THE UNIVERSITY OF WESTERN AUSTRALIA

Lecture 15
Lists to Dictionaries

Revision - Lists

- String and lists are subclasses of sequence
  - *Lists are mutable, but strings are not*
- Items in a list or string are obtained by indexing, with list (and string) items numbered from 0
- Lists can contain items of different types, e.g.

```
[1, 2.0, "three"]
```

- Lists are dynamic (they grow and shrink as required).

Sequence Operations

Operator	Meaning
<seq> + <seq>	Concatenation
<seq> * <int-expr>	Repetition
<seq>[]	Indexing
len(<seq>)	Length
<seq>[:]	Slicing
for <var> in <seq>:	Iteration
<expr> in <seq>	Membership (Boolean)

Sequence Operations

- Except for the membership check, we've used these operations before on strings.
- The membership operation can be used to see if a certain value appears anywhere in a sequence.

```
>>> lst = [1,2,3,4]
>>> 3 in lst
True
>>> month = 1
>>> year = 2000
>>> if month in [1,2] :  # month == 1 or month == 2
      year -= 1
```

List Operations

Method	Meaning
<list>.append(x)	Add element x to end of list.
<list>.sort()	Sort the list. A comparison function may be passed as a parameter. By default sorted in ascending order
<list>.reverse()	Reverse the list.
<list>.index(x)	Returns index of first occurrence of x.
<list>.insert(i, x)	Insert x into list at index i.
<list>.count(x)	Returns the number of occurrences of x in list.
<list>.remove(x)	Deletes the first occurrence of x in list.
<list>.pop(i)	Deletes the ith element of the list and returns its value.

List Operations

```
>>> lst = [3, 1, 4, 1, 5, 9]
>>> lst.append(2)
>>> lst
[3, 1, 4, 1, 5, 9, 2]
>>> lst.sort()
>>> lst
[1, 1, 2, 3, 4, 5, 9]
>>> lst.reverse()
>>> lst
[9, 5, 4, 3, 2, 1, 1]
>>> lst.index(4)
2
```

```
>>> lst.insert(4, "Hello")
>>> lst
[9, 5, 4, 3, 'Hello', 2, 1, 1]
>>> lst.count(1)
2
>>> lst.remove(1)
>>> lst
[9, 5, 4, 3, 'Hello', 2, 1]
>>> lst.pop(3)
3
>>> lst
[9, 5, 4, 'Hello', 2, 1]
```

List Operations

- Most of these methods don't return a value** – they change the contents of the list in some way.

```
>>> lst.sort()

>>> lst.sort(reverse=True)
```

- Lists can grow by appending new items, and shrink when items are deleted. Individual items or entire slices can be removed from a list using the `del` operator.

** They return None

List Operations

- ```
  >>> myList=[34, 26, 0, 10]
  >>> del myList[1]
  >>> myList
  [34, 0, 10]
  >>> del myList[1:3]
  >>> myList
  [34]
  ```

- `del` isn't a list method, but a built-in operation that can also be used on list items.
  ```

Tuples

- A *tuple* is a sequence which looks like a list but uses `()` rather than `[]`.
- Tuples are sequences that are **immutable**, so are used to represent sequences that are not supposed to change,
  - *e.g. student–mark pairs*
  - `[('Fred',55), ('Jemima',68), ('James',68)]`
  - *Sorting a list of tuples sorts on first member of each tuple*
  - *Turn a list into a tuple by using the* `tuple()` *function*

Dictionaries

- After lists, a dictionary is probably the most widely used collection/compound data type.
- Dictionaries are not as common in other languages as lists (arrays).
- Lists are sequential
  - *To find a particular need to search from the start.*
  - *Do you find a book in the library starting from Dewey number (000 is computer science!)*
    - Use catalogue!

Dictionaries

- Dictionaries use key-value pairs
- There are lots of examples!
  - *Names and phone numbers*
  - *Usernames and passwords*
  - *State names and capitals*
- A collection that allows us to look up information associated with arbitrary keys is called a mapping.
- Python dictionaries are *mapping*s. Other languages call them *hashes* or *associative arrays*.

Dictionaries

- Dictionaries can be created in Python by listing key–value pairs inside of curly braces.
- Keys and values are joined by `:` and are separated with commas.

```
>>>passwd = {"guido":"superprogrammer",
"turing":"genius", "bill":"monopoly"}
```

- We use an indexing notation to do lookups

```
>>> passwd["guido"]
'superprogrammer'
```

- Unlike list indexes, which are integers related to position in the list, dictionary indexes can be almost anything

Dictionaries

- `<dictionary>[<key>]` returns the object with the associated key.
- Dictionaries are mutable.

```
>>> passwd["bill"] = "bluescreen"

>>> passwd

{'guido': 'superprogrammer', 'bill':
'bluescreen', 'turing': 'genius'}
```

- Did you notice the dictionary printed out in a different order than it was created?

Initialising Dictionaries

- Dictionaries can be created directly

```
# dayno is in the range 0..6
dow_map = {0:"Sunday", 1:"Monday", 2:"Tuesday",
           3:"Wednesday”, 4:"Thursday", 5:"Friday",
           6:"Saturday"}
daystr = dow_map[dayno]
print(daystr)
```

Initialising Dictionaries

- Dictionaries can also be created incrementally. That is, start with an empty dictionary and add the key-value pairs one at a time.
- For example, assume the file `passwords` contains comma-separated pairs of user IDs and passwords

```
passwd_dir = {}
for line in open('passwords', 'r'):
    user, pw = line.strip().split(‘,’)
    passwd_dir[user] = pw
```

Dictionary Operations

Method	Meaning
<key> in <dict>	Returns true if dictionary contains the specified key, false if it doesn't.
<dict>.keys()	Returns a sequence of keys.
<dict>.values()	Returns a sequence of values.
<dict>.items()	Returns a sequence of tuples (key, value) representing the key-value pairs (i.e. 2-tuples).
del <dict>[<key>]	Deletes the specified entry.
<dict>.clear()	Deletes all entries.
for <var> in <dict>:	Loop over the keys.
<dict>.get(<key>, <default>)	If dictionary has key, returns its value; otherwise returns default.
<dict>[<key>]	If dictionary has key, return its value; otherwise exception raised

Dictionary Operations

```
>>> list(passwd.keys())
['guido', 'turing', 'bill']
>>> list(passwd.values())
['superprogrammer', 'genius', 'bluescreen']
>>> list(passwd.items())
[('guido', 'superprogrammer'), ('turing', 'genius'),
('bill', 'bluescreen')]
>>> "bill" in passwd
True
>>> "fred" in passwd
False
```

List of 2-tuples

Dictionary Operations

```
>>> passwd.get("guido", "unknown")
'superprogrammer'
>>> passwd.get("fred", "unknown")
'unknown'
>>> passwd["fred"]
Traceback (most recent call last):
  File "<pyshell>", line 1, in <module>
KeyError: 'fred'
>>> passwd.clear()
>>> passwd
{}
```

Example Program: Word Frequency

- We want to write a program that analyzes text documents and counts how many times each word appears in the document.
- This kind of analysis is sometimes used as a crude measure of the style similarity between two documents and is used by automatic indexing and archiving programs (like Internet search engines).

Example Program: Word Frequency

- This is a multi-accumulator problem!
- We need a count for each word that appears in the document.
- We can use a loop that iterates over each word in the document, incrementing the appropriate accumulator.
- The catch: we will likely need hundreds, perhaps thousands of these accumulators!

Example Program: Word Frequency

- Let's use a dictionary where strings representing the words are the keys and the values are `ints` that count up how many times each word appears.
  - *The mapping is: <string> → <int>*
- To update the count for a particular word, `w`, we need something like:

```
counts[w] += 1
```

- One problem – the first time we encounter a word it will not yet be in `counts`.

Example Program: Word Frequency

- Attempting to access a nonexistent key produces a run-time `KeyError`.

Pseudo-code

```
if w is already in counts:
   add one to the count for w
else:
   set count for w to 1
```

- How can this be implemented?

Example Program: Word Frequency

```
if w in counts:
    counts[w] += 1
else:
    counts[w] = 1
```

Can't do this in Python 2

- A more elegant approach:

```
counts[w] = counts.get(w, 0) + 1
```

- If `w` is not already in the dictionary, this `get` will return 0, and the result is that the entry for `w` is set to 1.

Example Program: Word Frequency

- The other tasks include
  - *Convert the text to lowercase (so occurrences of "Python" match "python")*
  - *Eliminate punctuation (so "python!" matches "python")*
  - *Split the text document into a sequence of words*

Example Program: Word Frequency

```
# get the sequence of words from the file
    fname = input("File to analyze: ")
    try:
      text = open(fname,'r').read()
    except IOError:
      print(“Cannot open {0:s}”.format(fname))
      return
    text = text.lower()
    for ch in '!"#$%&()*+,-./:;<=>?@[\\]^_`{|}~':
        text = text.replace(ch, ' ')
```

Example Program: Word Frequency

- Variable `text` has all the words in the file. Multiple spaces not a problem for `split()`

```
words = text.split()
```

- Loop through the words to build the counts dictionary

```
counts = {}

for w in words:

   counts[w] = counts.get(w,0) + 1
```

Example Program: Word Frequency

- print a list of words in alphabetical order with their associated counts

```
# get list of words that appear in document
# each word (i.e. key) is found only once!
uniqueWords = list(counts.keys())

# put list of words in alphabetical order
uniqueWords.sort()

# print words and associated counts
for w in uniqueWords:
    print(w, counts[w])
```

Example Program: Word Frequency

- This will probably not be very useful for large documents with many words that appear only a few times.
  - *Result will be a huge list*
- A more interesting analysis is to print out the counts for the *n* most frequent words in the document.
- To do this, we'll need to create a list that is sorted by counts (most to fewest), and then select the first *n* items.

Example Program: Word Frequency

- We can start by getting a list of key-value pairs using the `items` method for dictionaries.

  ```
  pairs = list(count.items())
  ```

- `pairs` will be a list of tuples like

  ```
  [('foo', 5), ('bar', 7), ('spam', 376)]
  ```

- If we try to sort them with `pairs.sort()`, they will be in ascending order of first component, i.e. dictionary order of the words.

  ```
  [('bar', 7), ('foo', 5), ('spam', 376)]
  ```

Example Program: Word Frequency

- Not what we wanted.
- To sort the items by frequency, we need a function that will take a tuple (here, 2-tuple) and return the second term, i.e. count.

```
def byCount(pair):
    return pair[1]
```

- To sort the list by frequency:

```
pairs.sort(key=byCount)
```

Example Program: Word Frequency

- We're getting there!
- What if have multiple words with the same number of occurrences? We'd like them to print in alphabetical order.
- That is, we want the list of pairs primarily sorted by count, but sorted alphabetically within each level.

Example Program: Word Frequency

- Looking at the documentation for sort, it says this method performs a "*stable* sort **in place**".
  - *"In place" means the method modifies the list that it is applied to, rather than producing a new list.*
  - *Stable means equivalent items (equal keys) stay in the same relative position to each other as they were in the original list.*

Example Program: Word Frequency

- If all the words were in alphabetical order before sorting them by frequency, words with the same frequency will be in alphabetical order!
- We just need to sort the list twice – first by words, then by frequency.

```
pairs.sort()                          # orders pairs alphabetically
pairs.sort(key = byCount, reverse = True) # orders by count
```

- Setting `reverse` to `True` tells Python to sort the list in reverse order.

Example Program: Word Frequency

- Now we are ready to print a report of the *n* most frequent words.
- Here, the loop index `i` is used to get the next pair from the list of items.
- That pair is unpacked into its `word` and `count` components.
- The word is then printed left–justified in fifteen spaces, followed by the count right–justified in five spaces.

Example Program: Word Frequency

```
for i in range(n):
    word, count = pairs[i]
    print("{0:<15}{1:>5}".format(word, count))
```

Example Program: Word Frequency

```
# A program to count word frequencies in text file
def byCount(pair):  # service function, select second of pair
    return pair[1]

def main():
    print("This program counts word frequency in a file and")
    print(”prints a report on the n most frequent words.\n")
    # get the sequence of words from the file
    fname = input("File to analyze: ")
    text = open(fname,'r').read()
    text = text.lower()
     for ch in '!"#$%&()*+,-./:;<=>?@[\\]^_`{|}~':
        text = text.replace(ch, ' ’)
```

Example Program: Word Frequency

```
words = text.split()

# construct a dictionary of word counts
counts = {}
for w in words:
    counts[w] = counts.get(w,0) + 1
# output analysis of n most frequent words.
n = int(input("Output analysis of how many words? "))
items = list(counts.items())  # word-count pair list
items.sort()   # alphabetic sort
items.sort(key=byCount, reverse=True)
for i in range(n):
    word, count = items[i]
    print("{0:<15}{1:>5}".format(word, count))
```

Summary

- We completed looking at Python lists, noting that many of the functions are actually methods that change the input list, esp. append and sort.
- We looked at tuples, as a special sort of list.
- We looked at dictionaries, as a mapping from keys to values which is not restricted to the order of items