THE UNIVERSITY OF MELBOURNE

Lecture 8: 2D Lists, Dictionaries and Sets.

Dr Simon D'Alfonso

School of Computing and Information Systems
Melbourne School of Engineering

Lecture 7 Challenges

- Write a function that devowels and returns an input string using list comprehension. Note that you will want to convert the input string to a list to do this, then reconstruct the devoweled string from the list comprehension result before returning it.
- Write a function that takes one tuple as input, returning True if all tuple items are of the same type, and False otherwise.

Object Identity

An interesting phenomenon was brought to our attention in last week's lecture via Slide 17. There seems to be an inconsistency with the way the *is* operator works, for example:

```
>>> int1 = 256
>>> int2 = 256
>>> int2 is int1
True

>>> int1 = 257
>>> int2 = 257
>>> int2 is int1
False
```

Why is this the case?

https://canvas.lms.unimelb.edu.au/courses/124575/discussion_topics/647420

MST Musings

Thinking about approaches to the types of questions found in the MST.

Solutions now available under LMS -> Modules -> Assessments

Today

1. 2-Dimensional Lists
2. Dictionaries, including Defaultdict
3. Sets

2-Dimensional Lists

- We have mainly been using lists in their standard, mono-dimensional structure, where lists contains things like integers, strings and floats.
  - lst1 = [1, 2, 3, 4]
  - lst2 = ['a', 'b', 'c', 'd’]
- Lists can also contain lists and other complex data types. Some example 2-D lists:
  - lst2d = [[1, 'a', 'first'], [2, 'b', 'second']]
  - lst_first_4 = [lst1, lst2]
- Lists can also contain lists that can contain lists and so on, but we’ll stop at the 2-D level

2-Dimensional Lists

To access a 2-D list element, use successive square brackets with the nested index positions:

```
>>> lst2d = [[0, 1, 2], ['zero', 'one', 'two']]
>>> lst2d[0][0]
0
>>> lst2d[0][2]
2
>>> lst2d[1][2]
'two'
```

2-Dimensional Lists

We can simply use nested *for* loops to successively go through each element of a 2-D list:

```
#collapse a 2-D list into a 1-D list
lst2d = [[1, 2, 3, 4, 5], [6, 7, 8, 9, 10]]
lst1d = []
for lst in lst2d:
    for item in lst:
        lst1d.append(item)
```

Using list comprehensions:

```
lst1d = [item for lst in lst2d for item in lst]
```

2-Dimensional Lists

- Row-column 2-dimensional tables are a very common data structure. E.g., Excel spreadsheets, websites (HTML: <table>, <tr>, <td>), etc.

Product Name	Quantity in Stock	Price	How Many Sold
Product1	5	45.5	10
Product2	8	10.2	50
Product3	9	6.4	20
Product4	20	8.6	20
Product5	15	10	30

- How can this be represented as a Python list structure?

2-Dimensional Lists

```
headers = ["Product Name", "Quantity in
Stock", "Price", "How Many Sold"]
prod1 = ["Product1", 5, 45.5, 10]
prod2 = ["Product2", 8, 10.2, 50]
prod3 = ["Product3", 9, 6.4, 20]
prod4 = ["Product4", 20, 8.6, 20]
prod5 = ["Product5", 15, 10, 30]
products = [headers, prod1, prod2, prod3,
prod4, prod5]


import pprint #pretty printer
pprint.pprint(products)
```

2-Dimensional lists

Change the price of Product1 to $50 and change the quantity of Product3 to 10:

```
products[0][2] = 50
products[2][1] = 10
print(products)
```

```
[['Product1', 5, 50, 10],
['Product2', 8, 10.2, 50],
['Product3', 10, 6.4, 20],
['Product4', 20, 8.6, 20],
['Product5', 15, 10, 30]]
```

2-Dimensional Lists

Print each row (i.e., product)

```
for i in range(len(products)):
    list_row = []
    for j in range(len(products[i])):
        list_row.append(products[i][j])

    print(list_row)
```

Print each column

```
for i in range(len(products[0])):
    list_column = []
    for j in range(len(products)):
        list_column.append(products[j][i])

    print(list_column)
```

Exercise 1

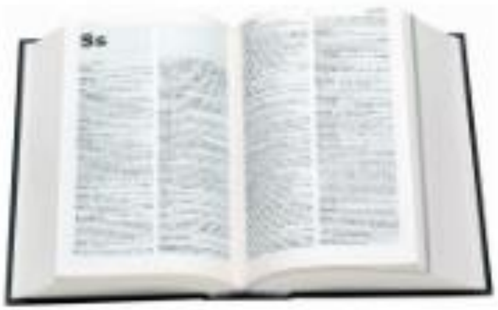
A valid table is one where each row has the same number of columns. Write a function table_valid(), which takes as input a 2-D list (the list represents a table of the form [row1, row2, row3]) and returns True or False depending on whether the table is valid.

Exercise 1 Solution

```
def table_valid(table):
    num_columns = len(table[0])
    for row in table:
        if len(row) != num_columns:
            return False

    return True
```

Dictionaries

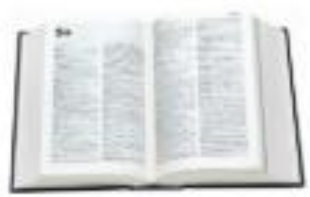


You know dictionaries!

They're great at looking up thing by a word, not a position in a list!

Look up

Hello



Get back

A greeting (salutation) said when meeting someone or acknowledging someone's arrival or presence.

Dictionaries

Unlike sequences, which are indexed by a range of numbers, dictionaries are indexed by unique *keys*, which can be any immutable type.

```
{key1:value1, key2:value2, key3:value3}
```

Using dictionaries

Dictionary initialization:

```
>>> drinks = {} #empty dictionary
>>> drinks = {"coffee": 3.00, "tea": 2.50}
>>> type(drinks)
<class 'dict'>
```

Iterating over a dictionary:

```
>>> for key in drinks:
        print(key)
coffee
tea
```

Looking up items in a dictionary:

```
>>> drinks["coffee"] #if not found then KeyError is raised
3.00
```

Adding or updating items to a dictionary:

```
>>> drinks["tea"] = 2.60
>>> drinks["water"] = 1.95
```

Dictionaries are mutable

Types for keys and values

- Keys in dictionaries (or elements in sets) must be **hashable**
- All of Python's immutable data types are **hashable**
  - **int, float, str, tuple, bool**
- **list, dict, set** cannot be keys in dictionaries (or elements in sets)
- Values in dictionaries can be of any of the data types we have been using, including dictionaries (i.e., dictionaries within dictionaries).

Dictionary Operations

- Is key in dictionary?
  - `<key_value> in dictionary`
- Deleting an entry and returning value of entry
  - `dictionary.pop(key)`
- Deleting the entire contents of a dictionary:
  - `dictionary.clear()`
- Deleting an entry (no return value)
  - `del dictionary[key]`
- Returning a dictionary sorted by key
  - `sorted_dictionary = sorted(dictionary)`

Exercise 2

Write a function lookup_capital(), that receives as input one string (a country name) and returns the capital city of that country.

There will be a global dictionary called CAPITALS that contains a collection of country names as keys with their capital city as corresponding value.

Exercise 2 Solution

```
CAPITALS = {'Australia': 'Canberra', 'Italy': 'Rome',
'England': 'London', 'China': 'Beijing'}


def lookup_capital(country):

    global CAPITALS

    if country in CAPITALS:
        return CAPITALS[country]
    else:
        return "I don't know that capital"
```

Extracting information from dictionaries

```
fruits = {'apples': 5, 'oranges': 6, 'bananas': 3}

#Store a list of all keys in the dictionary
keys = [k for k in fruits]
#or
keys = list(fruits.keys()) #returns all keys in dictionary

#Store a list of all values in the dictionary
vals = [fruits[k] for k in fruits]
#or
vals = list(fruits.values()) #returns all values in dictionary

#Store a list of (key, value) tuples for all items in the
dictionary
pairs = [(k, fruits[k]) for k in fruits]
#or
pairs = list(fruits.items()) #returns all (key, value) pairs
in dictionary
```

Exercise 3

Write a function word_count2() that takes as input a string of text and returns a dictionary containing an occurrence count for each word in the text.

Defaultdict

- When using dictionaries as “counters” or “accumulators” you need to initialize every value for new keys.
- Alternatively, you can simplify things with **defaultdict**. The functionality of both dictionaries and defaultdict are almost same except for the fact that defaultdict never raises a KeyError. It provides a default value for keys that do not exist yet.

```
from collections import defaultdict

def count_digits(num):
    # Count the occurrences of individual digits in a number
    digit_count = defaultdict(int) #default is the integer 0
    for digit in str(num):
        digit_count[digit] += 1

    return digit_count

print(count_digits(134345547343))
```

Sets

- Sets are unordered collections (not sequences), and their elements are unique (i.e., a set representation does not contain duplicates)
- They're good when you only want to store one of each thing but don't care about the order
- They are defined with curly brackets {}:
  - {1, 2, 3} is a set, and it is the same set as {3, 2, 1, 3}
- Kind of like just the keys part of dictionaries
- Set elements must be hashable/immutable
- Side fact: Sets are used in a branch of mathematics called set theory, and have been used to construct foundations for mathematical systems: https://en.wikipedia.org/wiki/Set_theory

Sets

```
>>> a = {1, 2, 3}
>>> b = {3, 2, 3, 1}
>>> a == b
True

>>> str_set = set('hannah') #a quick way to remove duplicates too
>>> str_set
{'h', 'a', 'n'}

>>> dict_set = set({'a': 1, 'b': 2})
>>> dict_set
{'b', 'a'}
```

Sets

```
>>> str_set = {'h', 'a', 'n'}
>>> 'h' in str_set # testing for membership
True
```

Remember that defining an empty dictionary is:

```
dict_name = {}
```

But {} also define a set. So how do we define an empty set?

```
a = set()
```

Sets

- Sets are not sequences (no order) so you can’t slice or index on them
- BUT they are mutable with methods add() and remove()

```
>>> a = set('cat')
>>> a.add('a')
{'t', 'a', 'c'} #didn't get added as letter 'a' already in set

>>> a.add('s')
{'t', 's', 'a', 'c'}

>>> a.remove('a')
{'t', 's', 'c'}
```

Sets

- The following operations help to illustrate the point of sets. Suppose we had the following two lists, which recorded city weather temperatures for a week:
  - melb_temps = [21, 25, 28, 19, 19, 25, 20]
  - syd_temps = [21, 30, 30, 18, 19, 27, 20]
- If we want to find the unique number of temperatures for each city, we can easily convert these lists to sets:
  - len(set(melb_temps)) == 5
  - len(set(syd_temps)) == 6
- Now, if we want to find the set of temperatures the cities had in common, we can use a set theory operation called **intersection**, which can be done in Python in two ways:
  - set(melb_temps) & set(syd_temps) == {19, 20, 21}
  - set(melb_temps).intersection(set(syd_temps)) {19, 20, 21}

Sets

- **Union** is another set theory operation that combines all the elements in two sets:
  - set(melb_temps) | set(syd_temps) == {18, 19, 20, 21, 25, 27, 28, 30}
  - set(melb_temps).union(set(syd_temps)) == {18, 19, 20, 21, 25, 27, 28, 30}

- **Set Difference** finds all the elements that are in one set but not the other:
  - set(melb_temps) - set(syd_temps) == {25, 28}
  - set(melb_temps).difference(set(syd_temps)) == {25, 28}

  - set(syd_temps) - set(melb_temps) == {18, 27, 30}
  - set(syd_temps).difference(set(melb_temps)) == {18, 27, 30}

Exercise 4

Given two sets, their symmetric difference is the set of elements that belong to either one or the other set but not both.

Write a function symmetric_difference(), that takes two sets as input and returns the set that is their symmetric difference.

Summary

Today we covered:

- Two-dimensional lists, which can be used to store tabular information.
- Dictionaries, which store items as **key: value** pairs
- For easy tallying of items use **defaultdict**
- **Sets** used for storing and manipulating collections with unique elements
- Dictionaries and sets are unordered collections that are **mutable**
- Dictionaries and sets use **hashing** to allow for efficient storage and retrieval without the use of indexes. Only immutable variable types are **hashable**

Lecture 8 Challenges

- Write a function swap_dict(), which takes one dictionary as input and returns another dictionary resulting from swapping the keys and values around. For example, if the input is {'a':1, 'b':2}, the output would be {1:'a', 2:'b'}.
- Write a function cartesian_product(), which takes two sets, A and B, as inputs. The function then returns the Cartesian product (x) of A and B. For example, if A = {1,2} and B = {3,4}, then A x B = {(1,3), (1,4), (2,3), (2,4)}

Lecture Identification and Acknowledgement

Coordinator / Lecturer: Simon D'Alfonso

Semester: Semester 1, 2022

These slides include materials from 2020 - 2021 instances of COMP90059 run by Kylie McColl or Wally Smith