

Lecture 7: Collections and their methods (Advanced Sequences & Mutability)

COMP90059 Introduction to Programming

Wally Smith

School of Computing & Information Systems

ASSESSMENT - REVISED on 9th April 2020

% contributions to overall grade for the subject are shown for each piece of assessment

- **Assignment 1** (10%) is ongoing and due on Friday 24th April.
- **Mid-Semester Test** (0%) will take place on Week 8, at 2 - 3 pm on Monday 4th May. This will be a one-hour test online via Grok in place of the usual Monday lecture. Note that your score on the test will NOT contribute to your overall grade for the subject, but it will provide you with feedback on your learning so far, and it will also give you a good indication of what the final examination will be like.
- **Assignment 2** (20%) is a second set of programming tasks on Grok. It will be released on Wednesday 6th May (during Week 8, after the Mid- Semester Test) , and is due on Friday 29th May (Week 11).
- **Final Examination** (70%) will be held in the examination period after semester (date to be advised). It will be delivered online in a way to be advised. You will carry out the exam during a specified 3 hours.
- **Hurdle requirement: 35/70 or greater** on the Final Examination is needed to pass the subject.

Please rest assured that the Final Examination is not intended to present fiendishly complex problems. Instead, it will present a mixture of questions that are designed to allow you demonstrate the knowledge and skills you have developed through the lectures, tutes and assignments. The nature of the Final Examination will be discussed in lectures in the remainder of semester, and we will look at many examples of the kinds of questions that will be included.

Lecture Overview

Coding Style

Consider some more style conventions from PEP8

Reviewing Lists and Functions

Consider the importance and value of lists and functions, by reworking the tennis scoring program from Lecture 3

Collections

Lists, Tuples, Dictionaries and Sets - consider what they are useful for

Mutability

A potentially tricky but important property of data structures in Python

Methods for collections

Looking in detail at some some important methods work to manipulate collections

Coding Style

Good practice for writing code

Style Guide of PEP8 (Python Enhancement Proposals)

<http://www.python.org/dev/peps/pep-0008>

- Code is read much more often than it is written
- Aim for greater consistency & readability

Some tips from PEP8

- Lines must not exceed 79 characters (use \ to indicate that the statement is continued on the next line)
- Use blank lines to separate logical sections (e.g. between functions and major sections of code associated with a given comment)
- 4 spaces per indent line level
- Variable names: lower case, meaningful names, camel casing (Lecture 2)
- Function names should be lowercase, with words separated by underscores
- Constants should be written in all capitals, with words separated by underscores (e.g., TAX_RATE) (Lecture 2)
- Don't compare Boolean values to True or False using ==

Reviewing Lists & Functions
- improving the tennis scoring
program

L3 Ex10: Scoring a game of tennis. Part 2

Write a program that takes in the number of points that two tennis players have scored against each other, and gives the game score so far.

Player A	Player B	Score
0	0	Love All
1	0	Fifteen - Love
1	2	Fifteen - Thirty
2	2	Thirty All
2	3	Thirty - Forty
2	4	Player B wins
3	3	Deuce
3	4	Advantage Player B
4	4	Deuce
4	5	Advantage Player B
6	5	Advantage Player A
7	5	Player A wins
9	11	Player B wins


```

#program to calculate tennis score V2
#V2 improves on V1 using lists and booleans
#V2 readability will be improved in later versions using functions
#V2 fails if impossible scores are entered - to be rectified in a later version

|

#take current number of points scored from user
pointsA = int(input('Enter number of As points: '))
pointsB = int(input('Enter number of Bs points: '))

#assign names for scores and points difference between players
lowScore = ['Love', 'Fifteen', 'Thirty', 'Forty', 'Player A Wins', 'Player B Wins']
deucePlusScore = ['Deuce', 'Advantage Player A', 'Player A Wins', 'Player B Wins', 'Advantage Player B']
pointsDifference = pointsA - pointsB

#produce message for scores of deuce or above
if pointsA >= 3 and pointsB >= 3:
    scoreMessage = deucePlusScore[pointsDifference]

#produce message for scores below deuce
if pointsA > 3:
    scoreMessage = lowScore[4]
elif pointsB > 3:
    scoreMessage = lowScore[5]
else:
    if pointsDifference == 0:
        scoreMessage = lowScore[pointsA] + ' All'
    else:
        scoreMessage = lowScore[pointsA] + ' - ' + lowScore[pointsB]

#report score message
print(scoreMessage)

```

#program to calculate tennis score V3

#V3 introduces functions which brings modularity with increased readability and traceability

#V3 fails if impossible scores are entered - to be rectified in a later version

```
def main():
```

```
    pointsA, pointsB = get_player_points()
```

```
    print(score_message(pointsA, pointsB))
```

```
def get_player_points():
```

```
    a = int(input('Enter number of A\'s points: '))
```

```
    b = int(input('Enter number of B\'s points: '))
```

```
    return a, b
```

```
def score_message(pointsA, pointsB):
```

```
    lowScore, deucePlusScore = define_scoreNames()
```

```
    pointsDifference = pointsA - pointsB
```

```
    if pointsA >= 3 and pointsB >= 3:
```

```
        return deucePlusScore[pointsDifference]
```

```
    else:
```

```
        if pointsA == 4:
```

```
            return lowScore[4]
```

```
        elif pointsB == 4:
```

```
            return lowScore[5]
```

```
        elif pointsDifference == 0:
```

```
            return lowScore[pointsA] + ' All'
```

```
        else:
```

```
            return lowScore[pointsA] + ' - ' + lowScore[pointsB]
```

```
def define_scoreNames():
```

```
    lowScore = ['Love', 'Fifteen', 'Thirty', 'Forty', 'Player A Wins', 'Player B Wins']
```

```
    deucePlusScore = ['Deuce', 'Advantage Player A', 'Player A Wins', 'Player B Wins', 'Advantage Player B']
```

```
    return lowScore, deucePlusScore
```

```
main()
```

Collections

The anatomy of programs (updated)

- **data structures** of different types,
 - variables:** strings, integers, floating points, booleans,
 - collections:** lists, tuples, sets, dictionaries

- **functions** - actions to be carried out on data
 - e.g., `print()`, `input()`, `abs()`, ... *and many more*
- **methods**
 - e.g., `<string>.split()`, `<list>.sort()` ... *and many more*

- **control flow** - redirecting the line-by-line flow of code
 - conditionals** (`if-elif-else`)
 - iteration** (`for` loops, `while` loops)

Collections

Collections are data structures that contain many data items.

There are four different kinds of collections shown below.

In other languages, collections are usually called 'arrays' and they are much less flexible than Python's collections.

- **lists**

ordered and **mutable** collections of values

- **tuples**

ordered and **immutable** collections of values

- **dictionaries**

un-ordered and **mutable** collections of **key-value pairs**

- **sets**


unordered collection of unique values that are **immutable**
(though the set overall is **mutable**)

Lists

A list is an **ordered** and **mutable** collection of values.

square brackets

```
customers = ['Johnson', 'Wang', 'Zhu', 'Agarwal', 'Williams', 'Nguyen']
```



* **assorted values:** can contain a mixture of types of data including collections

```
profile = [ 'Wilson', 45, True, [2, 4, 56, 23], 'Sydney', 3078]
```

* **ordered:** the list can be indexed and sliced

```
print(profile[1])           45
print(profile[2:4])         True, [2,4,56,23]
```

* **mutable:** items can be added or changed *in place*

```
profile[2] = 'hello'         ['Wilson', 45, 'hello', [2, 4, 56, 23], 'Sydney', 3078]
```

Lists are the most flexible and useful collections in Python.

Mutability allows lists to be built and modified dynamically during run-time.

Tuples

round brackets

A tuple is an **ordered** and **immutable** collection of values.

```
days = ('Monday', 'Tuesday', 'Wednesday', 'Thursday', 'Friday', 'Saturday', 'Sunday')
```

* **assorted types**: can contain a mixture of element types, including mutables (!)

```
days = ('Monday', 'Tuesday', 'Wednesday', 345, (2,3,4), [1,2,3], 'Sunday')
```

* **ordered**: can be indexed and sliced ...

```
print(day[2])           Wednesday
print(day[2:5])         ('Wednesday', 345, (2,3,4) )
```

square
brackets

... and can be used in a Boolean expression, and is iterable.

```
if 'Tuesday' in days:
```

```
for day in days:         MondayTuesdayWednesdayThursdayFridaySaturdaySunday
    print(day, end='')
```

* **immutable**: values cannot be changed in place

```
day[3] = 'Different'      TypeError: 'str' object does not support item assignment
```

A tuple is useful when we want to reinforce that the values cannot change.

#program to calculate tennis score V3

#V3 introduces functions which brings modularity with increased readability and traceability

#V3 fails if impossible scores are entered - to be rectified in a later version

```
def main():
```

```
    pointsA, pointsB = get_player_points()
```

```
    print(score_message(pointsA, pointsB))
```

```
def get_player_points():
```

```
    a = int(input('Enter number of A\'s points: '))
```

```
    b = int(input('Enter number of B\'s points: '))
```

```
    return a, b
```

```
def score_message(pointsA, pointsB):
```

```
    lowScore, deucePlusScore = define_scoreNames()
```

```
    pointsDifference = pointsA - pointsB
```

```
    if pointsA >= 3 and pointsB >= 3:
```

```
        return deucePlusScore[pointsDifference]
```

```
    else:
```

```
        if pointsA == 4:
```

```
            return lowScore[4]
```

```
        elif pointsB == 4:
```

```
            return lowScore[5]
```

```
        elif pointsDifference == 0:
```

```
            return lowScore[pointsA] + ' All'
```

```
        else:
```

```
            return lowScore[pointsA] + ' - ' + lowScore[pointsB]
```

```
def define_scoreNames():
```

```
    lowScore = ['Love', 'Fifteen', 'Thirty', 'Forty', 'Player A Wins', 'Player B Wins']
```

```
    deucePlusScore = ['Deuce', 'Advantage Player A', 'Player A Wins', 'Player B Wins', 'Advantage Player B']
```

```
    return lowScore, deucePlusScore
```

```
main()
```

These two collections would be better as tuples (with rounded brackets).

The immutability of a tuple will enforce the fact that they define given values and must not be changed by the program.

Dictionaries

braces, or
'curly brackets'

A dictionary is an **unordered** and **mutable** collection of **key-value pairs**.

```
stock_price = {'Lenovo' : 1999 , 'HP': 1750, 'Apple': 2950, 'Dell':1610}
```

* **assorted types**: there can be a mixture of types of data including other collections

```
stock = {'Lenovo' : [1999,230,235] , 'HP': 1750, 'Apple': 'expensive', 'Dell': {'screen':600, 'memory':700} }
```

* **unordered**: items are not in any order, so they cannot be indexed or sliced ...

... data in the dictionary is accessed through a key look up:

```
print(stock_price['HP'])
```

1750

```
print(stock_price[1750])
```

Error!! - look up only goes one way

... a dictionary is iterable:

```
for p in prices:  
    print(p)
```

Lenovo
HP
Apple
Dell

* **mutable**: items can be added or changed *in place*

```
stock_price['HP'] = 2019
```

```
print(stock_price)
```

{'Lenovo' : 1999 , 'HP': 2019, 'Apple': 2950, 'Dell':1610}

Dictionaries are the next most useful collection after lists.

square
brackets

Exercise 1: A password dictionary

Write a program using a dictionary to store the values of people's passwords shown in the table below. Allow user to enter a person's name, and the program will provide their password. For simplicity, we have just three users.

person	password
James	james2002
Julia	rover
Jacky	paswud

Sets

A set is an **unordered** and collection of unique values.

```
fruits= {'apple', 'banana', 'pear', 'orange', 'tomato', 'lemon'}
```

* assorted values: there can be a mixture of types of data, but not mutables

```
fruits= {'apple', 234, 'pear', ('onion', 'garlic'), 'tomato'} but not ['a', 'b', 'c']
```

* unordered: items cannot be indexed or sliced like a list.

```
print(fruits[1])  
print(fruits[3:5])
```

TypeError: 'set' object is not subscriptable

.... but can be used for booleans and iteration

```
for f in fruits:  
    print(f)
```

apple	lemon
pear	tomato
orange	orange
tomato	pear
banana	banana
lemon	apple

* immutable items: items cannot be changed *in place*

```
fruits[3] = 'cabbage'
```

TypeError: 'set' object does not support item assignment

A set is useful when we want to reinforce that values cannot change & there is NO order.

Exercise 2: Which kind of collection is likely to best?

Monday, Tuesday, Wednesday, Thursday, Friday, Saturday, Sunday

Kate (aged 80), Antonio (aged 91), Oscar (aged 88), Beverly (aged 83)

Queensland, New South Wales, Western Australia, Victoria, Tasmania,
South Australia, Northern Territory, Australian Capital Territories

Ashleigh Barty, Naomi Osaka, Caroline Wozniacki,
Karolína Plíšková, Simona Halep

Mutability

Mutability

Mutability is the property of a data type that determines if its value(s) can be changed *in place*.

mutables: lists, dictionaries

immutables: integers, floats, strings, Booleans, tuples

Lists are **mutable** ...

```
tennis_players = [Caroline Wozniacki, Karolína Plíšková, Simona Halep]  
tennis_players[0] = 'Ashleigh Barty' # this works fine
```

Tuples are **immutable** ...

```
day_names = (Monday, Tuesday, Wednesday, Thursday, Friday, Saturday, Sunday)  
day_names[4] = 'Vendredi' # this DOES NOT WORK
```

Strings are **immutable** ...

```
player = 'Karolína Plíšková'  
player[3] = 'o' # this DOES NOT WORK
```

Hold on !! Aren't strings and integers changeable?

It seems like we can change strings, integers, (& floats, Booleans) ...

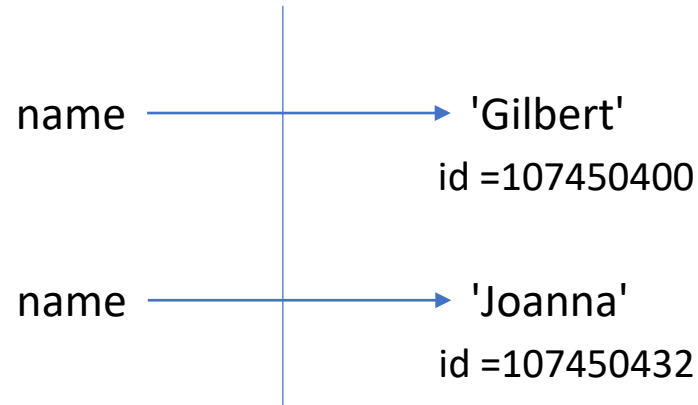
```
name = 'Gilbert'  
print(name)  
name = 'Joanna'  
print(name)
```

Gilbert
Joanna

To understand why this does not count as *changing in place* ...
... we have to think about what happens behind the scenes in Python

```
name = 'Gilbert'  
print(id(name))
```

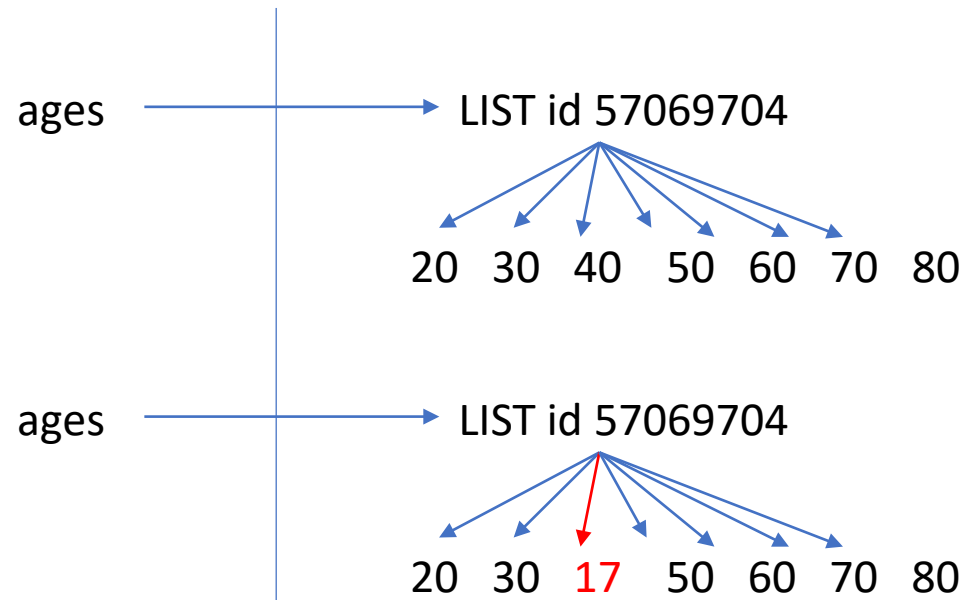
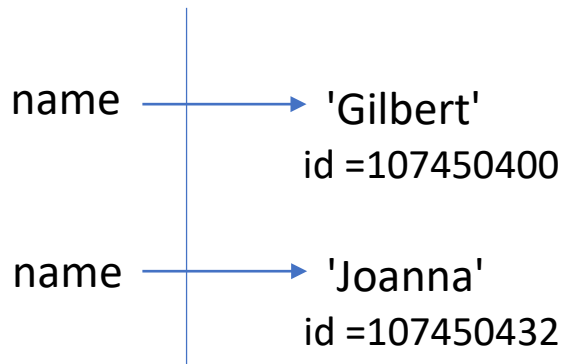
```
name = 'Joanna'  
print(id(name))
```



HOWEVER, if we change the values within a list the pattern is different ...

```
ages = [20,30,40,50,60,70,80]
print(ages, id(ages))           [20, 30, 40, 50, 60, 70, 80]  57069704
ages[2] = 17
print(ages, id(ages))           [20, 30, 17, 50, 60, 70, 80]  57069704
```

This is now a *change in place* ... because the underlying object is the same.



Why does mutability matter: aliasing

One important consequence of the mutability of lists is aliasing.
Assigning a list to another list, does NOT create a new list object ...
.. it just creates an ALIAS - a second name for the same underlying list object

```
ages = [20,30,40,50,60,70,80]
specials = ages
ages[2] = 17
print(ages, id(ages))           [20, 30, 17, 50, 60, 70, 80]  99255944
print(specials, id(specials))   [20, 30, 17, 50, 60, 70, 80]  99255944
```

This does NOT happen for immutables ...

```
age = 20
special = age
age = 17
print(age, id(age))
print(special, id(special))
```

```
17    1437853872
20    1437853920
```

```
age = 'young'
special = age
age = 'not so young'
print(age, id(age))
print(special, id(special))
```

```
not so young  96272496
young         99324000
```

Why does mutability matter: lists as parameters

Another important consequence of mutability ... i

what happens to **lists** when they are passed to functions as parameters

```
def main():
```

```
    ages = [20,30,40,50,60,70,80]
```

```
    print(ages, id(ages))
```

```
[20, 30, 40, 50, 60, 70, 80]
```

```
83218568
```

```
    double_values(ages)
```

```
    print(ages, id(ages))
```

```
[40, 60, 80, 100, 120, 140, 160]
```

```
83218568
```

```
def double_values(x):
```

```
    for i in range(len(x)):
```

```
        x[i] = 2 * x[i]
```

This does not happen to immutables:

integers, strings, floats, booleans or tuples

```
main()
```

```
def main():
```

```
    age = 20
```

```
    print(age, id(age))
```

```
20 1437853920
```

```
    double_values(age)
```

```
    print(age, id(age))
```

```
20 1437853920
```

```
def double_values(x):
```

```
    x = 2 * x
```

```
main()
```

Methods for Collections

List methods

method	description
<code><list>.append(x)</code>	add element x to end of list
<code><list>.sort()</code>	sorts the list into order
<code><list>.reverse()</code>	reverses elements in the list
<code><list>.index(x)</code>	returns the index of first occurrence of x in the list
<code><list>.insert(i, x)</code>	inserts x into the list at index i
<code><list>.count(x)</code>	returns the number of occurrences of x in the list
<code><list>.remove(x)</code>	deletes the first occurrence of x in the list
<code><list>.pop(i)</code>	deletes the i-th element in the list and returns its value (default removes last item in the list)
<code><list>.clear()</code>	remove all items from the list

list.append()

Adds a new value on to the end of a list - allowing us to build a list at run-time.

```
customers = ['Johnson', 'Wang', 'Zhu', 'Nguyen']
```

```
customers.append('Trudeau')
```

```
print(customers)
```

```
customers = ['Johnson', 'Wang', 'Zhu', 'Nguyen', 'Trudeau']
```

Exercise 3. Write a program to take in 10 new customer names from the user and add them into a growing list.

list.insert(i, x)

Inserts value x into the list at index i - it goes in before the current item at position i

```
customers = ['Johnson', 'Wang', 'Zhu', 'Nguyen']
```

```
customers.insert(2, 'Watson')
```

```
print(customers)
```

```
['Johnson', 'Wang', 'Watson', 'Zhu', 'Nguyen']
```

```
customers.insert(len(customers), 'Williams')
```

```
print(customers)
```

```
['Johnson', 'Wang', 'Watson', 'Zhu', 'Nguyen', 'Williams']
```

```
customers.insert(-1, 'Trudeau')
```

```
print(customers)
```

```
['Johnson', 'Wang', 'Watson', 'Zhu', 'Nguyen', 'Trudeau', 'Williams']
```

list.sort()

Sorts the list into a natural order - usually numerical or Unicode character order

```
customers = ['Johnson', 'Wang', 'Zhu', 'Nguyen', 'Trudeau']  
refunds = [123.34, 22.00, 3455.00, 16.00, 173.50]
```

```
customers.sort()  
print(customers)
```

['Johnson', 'Nguyen', 'Trudeau', 'Wang', 'Zhu']

```
refunds.sort()  
print(refunds)
```

[16.0, 22.0, 123.34, 173.5, 3455.0]

list.reverse()

Reverses the order of the list.

```
customers = ['Johnson', 'Wang', 'Zhu', 'Nguyen', 'Trudeau']  
customers.reverse()  
print(customers)
```

['Trudeau', 'Nguyen', 'Zhu', 'Wang', 'Johnson']

list.pop(i)

Deletes and returns the item in the list at position i

```
customers = ['Johnson', 'Wang', 'Zhu', 'Nguyen', 'Trudeau']

customers.pop(2)
print(customers)          ['Johnson', 'Wang', 'Nguyen', 'Trudeau']

candidate = customers.pop(0)
print(candidate)          'Johnson'
print(customers)          ['Wang', 'Nguyen', 'Trudeau']

customers.pop()
print(customers)          ['Wang', 'Nguyen']
```

list.remove(x)

Deletes the first occurrence of x in the list

```
items = ['kettle', 'dishwasher', 'dishwasher', 'toaster', 'grill', 'kettle', 'bbq', 'bbq']

items.remove('dishwasher')
print(items)
['kettle', 'dishwasher', 'toaster', 'grill', 'kettle', 'bbq', 'bbq']
```

list.clear()

Empties the list.

```
customers.clear()
print(customers)      []
```


list.index(x)

Returns the index of first occurrence of x in the list

```
customers = ['Johnson', 'Wang', 'Zhu', 'Nguyen', 'Trudeau']
```

```
position = customers.index('Wang')  
print(position)                                1
```

```
print(customers.index('Wang'))                  1
```

list.count(x)

Returns the number of occurrences of x in the list

```
itemsSold = ['kettle', 'dishwasher', 'dishwasher', 'toaster', 'grill', 'kettle', 'bbq',  
'bbq', 'cutlery', 'oven', 'freezer', 'bbq', 'oven', 'toaster', 'bbq', 'dishwasher']
```

```
print(itemsSold.count('bbq'))                    4
```

Exercise 4. Sales Data

Write a program to take a list of sold items and print out a list of the different items that shows how many of each have been sold.

```
itemsSold = ['kettle', 'dishwasher', 'dishwasher', 'toaster', 'grill', 'kettle',  
'bbq', 'bbq', 'cutlery', 'oven', 'freezer', 'bbq', 'oven', 'toaster', 'bbq',  
'dishwasher']
```

Exercise 5. Sales Data Dictionary

Modify the previous program so that it creates a dictionary called 'report' that contains each unique item paired with its number sold.

```
itemsSold = ['kettle', 'dishwasher', 'dishwasher', 'toaster', 'grill', 'kettle', 'bbq',  
'bbq', 'cutlery', 'oven', 'freezer', 'bbq', 'oven', 'toaster', 'bbq', 'dishwasher']  
report = {}
```

Functions that do the same or similar things ...

A confusing thing in Python is that functions and methods often to similar things.

```
values =[10,40,20,15]
```

```
for i in reversed(values):  
    print (i)
```

15
20
40
10

```
for i in sorted(values):  
    print (i)
```

10
15
20
40

```
print(min(values), max(values))
```

10 40

Dictionary methods

method	description
<code><key> in <dict></code>	returns true if dictionary contains the specified key, false if it doesn't
<code><dict>.keys()</code>	returns a sequence of keys
<code><dict>.values()</code>	returns a sequence of values
<code><dict>.items()</code>	returns a sequence of tuples (key, value) of key-value pairs
<code><dict>.get(<key>, <default>)</code>	if dictionary has <key> returns its value, else returns <default>
<code>del <dict>[<key>]</code>	deletes the specified key-pair entry
<code><dict>.clear()</code>	deletes all entries
<code>for <var> in <dict></code>	loops through the key values

```
sales = {'kettle': 2, 'dishwasher': 3, 'toaster': 2, 'grill': 1, 'bbq': 4, 'cutlery': 1, 'oven': 2, 'freezer': 1}
```

```
print('dishwasher' in sales)    True
print('fridge' in sales)       False
```

```
print(sales.keys())            ['kettle', 'dishwasher', 'toaster', 'grill', 'bbq', 'cutlery', 'oven', 'freezer']
```

```
print(sales.values())          [2, 3, 2, 1, 4, 1, 2, 1]
```

```
print(sales.items())
[ ('kettle', 2), ('dishwasher', 3), ('toaster', 2), ('grill', 1), ('bbq', 4), ('cutlery', 1), ('oven', 2), ('freezer', 1) ]
```

```
print(sales.get('bbq','not present'))    4
print(sales.get('fridge','not present'))  not present
```

```
del sales['kettle']
print(sales)
{'dishwasher': 3, 'toaster': 2, 'grill': 1, 'bbq': 4, 'cutlery': 1, 'oven': 2, 'freezer': 1}
```

```
sales.clear()
print(sales)    { }
```

Challenge

Write a program that can print out hands of cards from a 'shuffled' pack of playing cards.

```
***** Round 1 *****  
Player 1 : 9♥ 4♥ 3♥ 2♦ 10♠  
Player 2 : J♣ 7♦ K♦ 6♠ K♣  
Player 3 : 7♠ 7♣ 9♦ K♠ 5♣  
Player 4 : 8♦ 10♣ J♦ Q♠ 8♠  
Player 5 : 5♠ 4♣ 5♦ 8♥ 2♣  
Player 6 : 6♦ A♣ Q♥ 8♣ J♠  
Player 7 : 4♠ 4♦ 3♣ Q♣ 5♥  
Player 8 : Q♦ 10♥ A♦ 9♣ 2♠
```

```
suitNames = ('♣','♦','♥','♠')  
valueNames = ('A','2','3','4','5','6','7','8','9','10','J','Q','K')
```

Random function

The random library is useful for generating pseudo-random values.

```
import random

for i in range(10):
    print(random.randint(1, 10))

for card in range(52):
    suit = random.randint(0,3)
    value = random.randint(0,13)
    card = valueNames[value] + suitNames[suit]

suitNames = ('♣', '♦', '♥', '♠')
valueNames = ('A','2','3','4','5','6','7','8','9','10','J','Q','K')
```


manufacture(deck) - round 1

['A♣', '2♣', '3♣', '4♣', '5♣', '6♣', '7♣', '8♣', '9♣', '10♣', 'J♣', 'Q♣', 'K♣', 'A♦', '2♦', '3♦', '4♦', '5♦', '6♦', '7♦', '8♦', '9♦', '10♦', 'J♦', 'Q♦', 'K♦', 'A♥', '2♥', '3♥', '4♥', '5♥', '6♥', '7♥', '8♥', '9♥', '10♥', 'J♥', 'Q♥', 'K♥', 'A♠', '2♠', '3♠', '4♠', '5♠', '6♠', '7♠', '8♠', '9♠', '10♠', 'J♠', 'Q♠', 'K♠']

shuffle(deck) - round 1

['K♠', 'K♥', '6♠', '7♠', 'Q♣', '7♦', '3♦', '8♣', '4♦', '9♠', '4♥', '4♣', 'A♥', 'J♥', '2♥', '7♣', '10♣', 'K♣', 'K♦', 'J♣', '8♦', '5♠', '2♠', 'Q♠', 'A♠', '3♠', '8♠', '5♦', '4♠', '6♣', '7♥', '6♥', '9♦', '10♠', '10♥', '2♦', 'A♣', '8♥', 'Q♦', '2♣', '10♦', '6♦', '9♣', 'Q♥', '5♥', '9♥', 'A♦', '3♣', '5♣', '3♥', 'J♠', 'J♦']

***** Round 1 *****

Player 0 : K♠ K♥ 6♠ 7♠ Q♣
Player 1 : 7♦ 3♦ 8♣ 4♦ 9♠
Player 2 : 4♥ 4♣ A♥ J♥ 2♥
Player 3 : 7♣ 10♣ K♣ K♦ J♣
Player 4 : 8♦ 5♠ 2♠ Q♠ A♠
Player 5 : 3♠ 8♠ 5♦ 4♠ 6♣
Player 6 : 7♥ 6♥ 9♦ 10♠ 10♥
Player 7 : 2♦ A♣ 8♥ Q♦ 2♣

***** Round 1 *****

Player 0 : 9♥ 4♥ 3♥ 2♦ 10♠
Player 1 : J♣ 7♦ K♦ 6♠ K♣
Player 2 : 7♠ 7♣ 9♦ K♠ 5♣
Player 3 : 8♦ 10♣ J♦ Q♠ 8♠
Player 4 : 5♠ 4♣ 5♦ 8♥ 2♣
Player 5 : 6♦ A♣ Q♥ 8♣ J♠
Player 6 : 4♠ 4♦ 3♣ Q♣ 5♥
Player 7 : Q♦ 10♥ A♦ 9♣ 2♠

***** Round 3 *****

Player 0 : J♦ A♣ 6♠ 2♠ J♠
Player 1 : 9♠ Q♦ 3♥ J♣ 8♣
Player 2 : 4♠ 7♣ 3♦ K♦ 5♦
Player 3 : 9♣ 8♦ 4♥ 2♥ 10♥
Player 4 : J♥ 6♦ 10♣ 4♣ 3♣
Player 5 : 5♠ 9♦ 2♣ 5♣ 7♥
Player 6 : 8♥ A♥ 10♠ 6♥ 10♦
Player 7 : 4♦ A♦ 7♠ 8♠ 3♠