Mutable and Immutable Objects

Lecture 7

Department of Computer Science and Engineering, ITER
Siksha ‘O’ Anusandhan (Deemed to be University), Bhubaneswar, Odisha, India.

Contents

Introduction

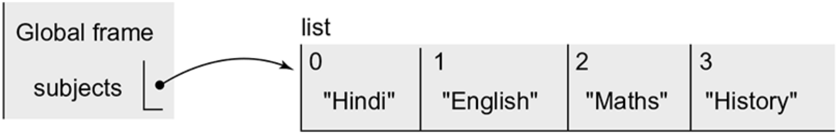
- In Python, ‘mutable’ is the ability of objects to change their values.
- In Python, if the value of an object cannot be changed over time, then it is known as immutable.
- List, Set and Dictionary are mutable.
- Strings and Tuples are immutable.
- Elementary forms of data such as numeric and Boolean are called scalar data types.
- Several applications require more complex forms of data, for example, the name of a person, coordinates of a point, a set of objects, or a list of personal records of individuals.

Lists

- A list is an ordered sequence of values. It is a non-scalar type.
- Values stored in a list can be of any type such as string, integer, float, or list, for example, a list may be used to store the names of subjects:
  ⟩⟩⟩ subjects=['Hindi', 'English', 'Maths', 'History']
- The elements of a list are enclosed in square brackets, separated by commas.
- Elements of a list are arranged in a sequence beginning index 0, just like characters in a string.

Lists (Cont.)

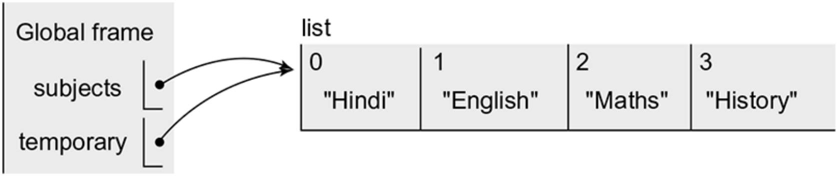
- We show the representation of the list subjects as in PythonTutor.



- To understand the lists better, let us invoke the function id that returns object identifier for the list object subjects:
  ⟩⟩⟩ id(subjects)
  57135752
- Different variables may refer to the same list object
  ⟩⟩⟩ temporary = subjects
  ⟩⟩⟩ id(temporary)
  57135752

Lists (Cont.)

- Each of the names subjects and temporary is associated with the same list object having object id 57135752.
- PythonTutor representation of the lists subjects and temporary, at this point, is shown below:



- This method of accessing an object by different names is known as aliasing.

Lists (Cont.)

- As each of the two names temporary and subjects refers to the same list, on modifying the component temporary[0] of the list temporary, the change is reflected in subjects[0] as shown below:

```
⟩⟩⟩ temporary[0] = 'Sanskrit'
⟩⟩⟩print(temporary)
['Sanskrit', 'English', 'Math', 'History']
⟩⟩⟩ print(subjects)
['Sanskrit', 'English', 'Math', 'History']
⟩⟩⟩ print(id(subjects), id(temporary))
57135752    57135752
```

Lists (Cont.)

- **Two-dimensional list: list of lists**
- we create a list of subjects and their corresponding subject codes. For this purpose, we represent each pair of subject and subject code as a list, and form a list of such lists:

  ```
  ⟩⟩⟩ subjectCodes = [['Sanskrit', 43], ['English', 85] , ['Maths', 65],
  ['History', 36]]
  ```
- A list of lists such as subjectCodes, each of whose elements itself is a list, is called a two-dimensional list.
- Thus, subjectCodes[1] being a list, its components may be accessed as subjectCodes[1][0] and subjectCodes[1][1]:

  ```
  ⟩⟩⟩ subjectCodes[1]
  ['English', 85]
  ⟩⟩⟩ print(subjectCodes[1][0], subjectCodes[1][1])
  English 85
  ```

Lists (Cont.)

- Often times, we need to take a list as an input from the user. For this purpose, we use the function **input** for taking the input from the user, and subsequently apply the function **eval** for transforming the raw string to a list:

```
>>> details = eval(input('Enter details of Megha: '))
Enter details of Megha: ['Megha Verma', 'C-55, Raj Nagar,Pitam Pura, Delhi - 110034', 9876543210]
>>> details
['Megha Verma', 'C-55, Raj Nagar,Pitam Pura, Delhi - 110034', 9876543210]
```

Lists (Cont.)

- we describe some functions and operations on lists with the help of the following lists:
  ⟩⟩⟩ list1 = ['Red', 'Green']
  ⟩⟩⟩ list2 = [10, 20, 30]

Lists (Cont.)

Operation	**Example**
Multiplication Operator $*$	$\rangle\rangle\rangle$ list2 $*$ 2 [10, 20, 30, 10, 20, 30]
Concatenation Operator $+$	$\rangle\rangle\rangle$ list1=list1 $+$ ['Blue'] $\rangle\rangle\rangle$ list1 ['Red', 'Green', 'Blue']
Length Operator **len**	$\rangle\rangle\rangle$ len(list1) 3
Indexing	$\rangle\rangle\rangle$ list2[-1] 30
Slicing Syntax: start:end:inc	$\rangle\rangle\rangle$ list2[0:2] [10, 20] $\rangle\rangle\rangle$ list2[0:3:2] [10, 30]
Function **min**	$\rangle\rangle\rangle$ min(list2) 10

Lists (Cont.)

Operation	**Example**
Function **max**	⟩⟩⟩ max(list1) 'Red'
Function **sum**(Not defined on strings)	⟩⟩⟩ sum(list2) 60
Membership operator **in**	⟩⟩⟩ 40 in list2 False

Table 1: Summary of operations that can be applied on lists

Lists (Cont.)

- The membership operator **in** may be used in a for loop for sequentially iterating over each element in the list, for example: iterating over the elements of a list

```
⟩⟩⟩ students = ['Ram', 'Shyam', 'Gita', 'Sita']
⟩⟩⟩ for name in students:
        print(name)
Ram
Shyam
Gita
Sita
```

Lists (Cont.)

- We list some important built-in functions that can be applied to lists.
- Many of these functions such as **append, reverse, sort, extend, pop, remove,** and **insert** modify the list.
- Functions such as **count** and **index** do not modify the list.
- **dir(list)** outputs the list including all the functions that can be applied to objects of the type **list**.

Lists (Cont.)

Function	**Explanation**
L.append(e)	Inserts the element e at the end of list L.
L.extend(L2)	Inserts the item in the sequence L2 at the end of elements of the list L.
L.remove(e)	Removes the first occurrence of the element e from the list L
L.pop(i)	Returns the elements from the list L at index i, while removing it from the list.
L.count(e)	Returns count of occurrences of object e in the list L.
L.index(e)	Returns index of an object e, if present in list.
L.insert(i,e)	Inserts element e at index i in list.
L.sort()	Sorts the elements of list.
L.reverse()	Reverses the order of elements in the list.

Table 2: List functions

Lists (List Comprehension)

- List comprehension provides a shorthand notation for creating lists.
- Suppose we want to create a list that contains cubes of numbers ranging from 1 to 10.
- To do this, we may create an empty list, and use a for-loop to append the elements to this list:

```
⟩⟩⟩ cubes =[]
⟩⟩⟩ end = 10
⟩⟩⟩for i in range(1, end + 1):
      cubes.append(i ** 3)
⟩⟩⟩ cubes
[1, 8, 27, 64, 125, 216, 343, 512, 729, 1000]
```

Lists (List Comprehension)

- Alternatively, a simple one line statement can be used for achieving the same task.
  ⟩⟩⟩ cubes = [x∗∗3 for x in range(1, end + 1)]

Lists (List as Arguments)

```python
def listUpdate(a, i, value):
    '''
    Objective: To change a value at a particular index in list
    Input Parameters:
    a - list
    i - index of the object in the list to be updated
    value - modified value at index i
    Return value: None
    '''
    a[i] = value
def main():
    '''
    Objective: To change a value at a particular index in list
    Input Parameters: None
    Return value: None
    '''
    lst = [10, 20, 30, [40, 50]]
    listUpdate(lst, 1, 15)
    print(lst)
if __name__=='__main__':
    main()
```

Lists (List as Arguments)

- The function listUpdate updates the list **a** by replacing the object a[i] by value.
- The main function invokes the function listUpdate with the arguments lst, 1, and 15 corresponding to the formal parameters a, i, and value.
- As arguments are passed by reference, during execution of the function listUpdate, an access to the formal parameter **a** means access to the list **lst** created in the main function.
- Consequently, when we update the list **a** in the function listUpdate, it results in the corresponding update of the list **lst**.
- Thus, the value at index 1 of the list **lst** gets updated to the value 15.

Lists (Copying List Objects)

- assigning a list to another name does not create another copy of the list, instead it creates another reference

```
⟩⟩⟩ list1 = [10, 20, [30, 40]]
⟩⟩⟩ list2 = list1
```

- As the names list1 and list2 refer to the same list object, any changes made in the list will relate to both the names list1 and list2, for example:

```
⟩⟩⟩ list1[1] = 22
⟩⟩⟩ list1
[10, 22, [30, 40]]
⟩⟩⟩ list2
[10, 22, [30, 40]]
```

Lists (Copying List Objects)

- To create another instance of the list object having different storage, we need to import the copy module and invoke the function **copy.copy()**:

```
〉〉〉 import copy
〉〉〉 list1 = [10, 20, [30, 40]]
〉〉〉 list3 = copy.copy(list1)
```

- the copy function creates a new copy list3 of the list1. Consequently, on modifying list1[1], list3[1] remains unaltered:

```
〉〉〉 list1[1] = 25
〉〉〉 list1
[10, 25, [30, 40]]
〉〉〉 list3
[10, 20, [30, 40]]
```

Lists (Copying List Objects)

- However, the copy function creates a shallow copy i.e. it does not create copies of the nested objects. Thus, the two lists share the same nested objects.
- list1[2] and list3[2] refer to the same nested list [30, 40].
- let us modify list1[2][0] in sub-list list1[2] of list list1 to value 35.

```
⟩⟩⟩ list1[2][0] = 35
⟩⟩⟩ list1
[10, 25, [35, 40]]
⟩⟩⟩ list3
[10, 20, [35, 40]]
```

Lists (Copying List Objects)

- To create a copy of a list object so that the nested objects (at all levels) get copied to new objects, we use the function deepcopy of the copy module:
- **deepcopy()**: To create a copy of a list including copies of the nested objects at all level

```
⟩⟩⟩ import copy
⟩⟩⟩ list1 = [10, 20, [30, 40]]
⟩⟩⟩ list4 = copy.deepcopy(list1)
⟩⟩⟩ list1[2][0] = 35
⟩⟩⟩ list1
[10, 20, [35, 40]]
⟩⟩⟩ list4
[10, 20, [30, 40]]
```

Lambda Expression

- A lambda expression consists of the lambda keyword followed by a comma separated list of arguments and the expression to be evaluated using the list of arguments in the following format:
- **lambda arguments: expression**
- Example: lambda expression to compute the cube of a number

```
>>> cube = lambda x: x ** 3
>>> cube(3)
27
```

- Example: lambda expression to compute sum of cubes of two numbers

```
>>> sum2Cubes = lambda x, y: x**3 + y**3
>>> sum2Cubes(2, 3)
35
```

map, reduce, and filter Operations on a Sequence

- Python provides several built-in functions based on expressions, which work faster than loop-based user defined code.
- The function map is used for transforming every value in a given sequence by applying a function to it.
- It takes two input arguments: the iterable object (i.e. object which can be iterated upon) to be processed and the function to be applied, and returns the map object obtained by applying the function to the list as follows:
- **result = map(function, iterable object)**
- The function to be applied may have been defined already, or it may be defined using a lambda expression which returns a **function** object.

map, reduce, and filter Operations on a Sequence

- Mapping every value in the sequence to its cube

```
⟩⟩⟩ lst = [1, 2, 3, 4, 5]
⟩⟩⟩ list(map(lambda x: x ** 3, lst))
[1, 8, 27, 64, 125]
⟩⟩⟩ list(map(cube, lst))
[1, 8, 27, 64, 125]
```

- We may also use any system defined or user-defined function as an argument of the map function, for example:

```
⟩⟩⟩ list(map(abs, [-1, 2, -3, 4, 5]))
[1, 2, 3, 4, 5]
```

map, reduce, and filter Operations on a Sequence

- Suppose we wish to compute the sum of cubes of all elements in a list.
- Adding elements of the list is a repetitive procedure of adding two elements at a time.
- The function reduce, available in functools module, may be used for this purpose. It takes two arguments: a function, and a iterable object, and applies the function on the iterable object to produce a single value:

```
>>> lstCubes = list(map(lambda x: x ** 3,lst))
>>> import functools
>>> sumCubes = functools.reduce(lambda x, y: x + y, lstCubes)
>>> sumCubes
225
```

map, reduce, and filter Operations on a Sequence

- To compute the sum of cubes of only those elements in the list lstCubes that are even.
- Thus, we need to **filter** the even elements from the list lstCubes
- Python provides the function **filter** that takes a function and a iterable object as the input parameters and returns only those elements from the iterable object for which function returns True.
- Since the function **filter** returns a **filter** object, we have used the function **list** to convert it to list.

map, reduce, and filter Operations on a Sequence

```
⟩⟩⟩ evenCubes = list(filter(lambda x: x%2== 0, lstCubes))
⟩⟩⟩ evenCubes
[8, 64]
⟩⟩⟩ sumEvenCubes = functools.reduce(lambda x, y: x + y,
evenCubes)
⟩⟩⟩ sumEvenCubes
72
```

- Alternatively, the sum of odd cubes may be computed as follows:

```
⟩⟩⟩ sumEvenCubes = functools.reduce(lambda x, y: x + y,
filter(lambda x: x%2 == 0, lstCubes))
⟩⟩⟩ sumEvenCubes
72
```

- The functions map, reduce, and filter are often used to exploit the parallelism of the system to distribute computation to several processors.

Sets

- A comma separated unordered sequence of values enclosed within curly braces is called **Set.**
- Function **set()** is used to convert a sequence to a set.

```
⟩⟩⟩ vowels = set('aeiou')
⟩⟩⟩ vowels
{'e', 'a', 'o', 'u', 'i'}
⟩⟩⟩ vehicles = set(['Bike', 'Car', 'Bicycle', 'Scooter'])
⟩⟩⟩ vehicles
{'Scooter', 'Car', 'Bike', 'Bicycle'}
⟩⟩⟩ digits = set((0, 1, 2, 3, 4, 5, 6, 7,8, 9))
⟩⟩⟩ digits
{0, 1, 2, 3, 4, 5, 6, 7, 8, 9}
```

Sets

- Elements of a set must be immutable.
- Set type does not support indexing, slicing, + operator, and * operator.
- We can iterate over elements of the set using **in** operator

```
⟩⟩⟩ for v in vowels:
        print(v, end = ' ')
e a o u i
```

- The functions **min, max, sum**, and **len** work for sets in the same manner as defined for lists.

```
⟩⟩⟩ len(vehicles)
4
```

- The membership operator **in** checks whether an object is in the set.

```
⟩⟩⟩ 'Bike' in vehicles
True
```

Sets (Functions)

- List of some important built-in functions that can be applied to sets.

Function	**Description**
S.add(e)	Adds the elements to the set S, if not present already.
S1.update(L1)	Add the items in object L1 to the set S1, if not already present.
S.remove(e)	Removes the element e from set S.
S.pop()	Removes an element from the set S.
S.clear()	Removes all elements from the set S.
S.copy()	Creates a copy of the set S.
S1.union(S2)	Returns union of the Sets S1 and S2.
S1.intersection(S2)	Returns a set containing common elements of sets S1 and S2.

Sets (Functions)

- List of some important built-in functions that can be applied to sets.

Function	**Description**
S1.difference(S2)	Returns a set containing elements in set S1 but not in set S2.
S1.symmetric_difference(S2)	Returns a set containing elements that are in one of the two sets S1 and S2, but not in both.

Table 3: Set Functions

- The operators $<=$, $==$, and $>=$ may be used to check whether a given set is a subset, equal, or superset of another set.

Sets (Set Comprehension)

- Set comprehension is applied to find common factors
- A common factor cannot exceed smaller of the two numbers, say n1 and n2.
- To build a set of common factors, we make use of comprehensions. For each number i in range(1, min(n1,n2) + 1), we include it in the set if it is a factor of each of n1 and n2.
- Thus, our code comprises just one line:

```
commonFactors = {i for i in range(1,min(n1+1, n2+1)) if n1%
    i == 0 and n2%i ==0}
```

Tuples

- A **tuple** is a non-scalar type defined in Python. Just like a list, a tuple is an ordered sequence of objects.
- However, unlike **lists**, **tuples** are immutable, i.e. elements of a **tuple** cannot be overwritten.
- A tuple may be specified by enclosing in the parentheses, the elements of the tuple (possibly of heterogeneous types), separated by commas, for example, the tuple t1 comprises five objects:

  ```
  ⟩⟩⟩ t1 = (4, 6, [2, 8], 'abc', {3,4})
  ⟩⟩⟩ type(t1)
  ⟨class'tuple'⟩
  ```

Tuples

- If a tuple comprises a single element, the element should be followed by a comma to distinguish a tuple from a parenthesized expression, for example:
  ```
  ⟩⟩⟩ (2,)
  (2,)
  ```
- A tuple having a single element is also known as singleton tuple.
- Another notation for tuples is just to list the elements of a tuple, separated by commas:
  ```
  ⟩⟩⟩ 2, 4, 6
  (2, 4, 6)
  ```
- Elements of a tuple may be mutable
  ```
  ⟩⟩⟩ t1 = (1, 2, [3, 4])
  ⟩⟩⟩ t1[2][1] = 5
  ⟩⟩⟩ t1
  (1, 2, [3, 5])
  ```

Tuple Operations

- we summarize the operations on tuples and use the following tuples t1 and t2 for the purpose of illustration:
  ⟩⟩⟩ t1 = ('Monday', 'Tuesday')
  ⟩⟩⟩ t2 = (10, 20, 30)

Operation	Example
Multiplication Operator $*$	⟩⟩⟩ t1 $*$ 2 ('Monday', 'Tuesday', 'Monday', 'Tuesday')
Concatenation Operator $+$	⟩⟩⟩ t3=t1 $+$ ('Wednesday',) ⟩⟩⟩ t3 ('Monday', 'Tuesday', 'Wednesday')
Length Operator **len**	⟩⟩⟩ len(t1) 2
Indexing	⟩⟩⟩ t2[-2] 20

Tuple Operations

Operation	**Example**
Slicing Syntax: start:end:inc	⟩⟩⟩ t1[1:2] ('Tuesday',)
Function **min**	⟩⟩⟩ min(t2) 10
Function **max**	⟩⟩⟩ max(t2) 30
Function **sum** (not defined on strings)	⟩⟩⟩ sum(t2) 60
Membership operator **in**	⟩⟩⟩ 'Friday' in t1 False

Table 4: Summary of operations that can be applied on tuples

Functions Tuple and Zip

- The function tuple can be used to convert a sequence to a tuple, for example:

```
⟩⟩⟩ vowels = 'aeiou'
⟩⟩⟩ tuple(vowels)
('a', 'e', 'i', 'o', 'u')
```

- The function zip is used to produces a zip object (iterable object), whose ith element is a tuple containing ith element from each iterable object passed as argument to the zip function.
- We have applied list function to convert the zip object to a list of tuples. For example,

```
⟩⟩⟩ colors = ('red', 'yellow', 'orange')
⟩⟩⟩ fruits = ['cherry', 'banana', 'orange']
⟩⟩⟩ fruitColor = list(zip(colors, fruits))
⟩⟩⟩ fruitColor
[('red', 'cherry'),('yellow', 'banana'),('orange', 'orange')]
```

Functions count and index

- The function **count** is used to find the number of occurrences of a value in a tuple, for example:
  ```
  ⟩⟩⟩ age = (20, 18, 17, 19, 18, 18)
  ⟩⟩⟩ age.count(18)
  ```
- The function **index** is used to find the index of the first occurrence of a particular element in a tuple, for example:
  ```
  ⟩⟩⟩ age.index(18)
  1
  ```

Function	**Explanation**
T.count(e)	Returns count of occurrences of e in Tuple T
T.index(e)	Returns index of first occurrences of e in Tuple T

Table 5: Tuple Functions

Dictionary

- Unlike lists, tuples, and strings, a **dictionary** is an unordered sequence of key-value pairs.
- Indices in a **dictionary** can be of any immutable type and are called keys.
- Beginning with an empty dictionary, we create a dictionary of month_number-month_name pairs as follows:

```
>>> month = {}
>>> month[1] = 'Jan'
>>> month[2] = 'Feb'
>>> month[3] = 'Mar'
>>> month[4] = 'Apr'
>>> month
{1: 'Jan', 2: 'Feb', 3: 'Mar', 4: 'Apr'}
>>> type(month)
<class'dict'>
```

Dictionary

```
⟩⟩⟩ price = {'tomato':40, 'cucumber':30, 'potato':20,
'cauliflower':70, 'cabbage':50, 'lettuce':40, 'raddish':30, 'carrot':20,
'peas':80}
⟩⟩⟩ price['potato']
20
⟩⟩⟩ price['carrot']
20
⟩⟩⟩ price.keys()
dict_keys(['tomato', 'cucumber', 'potato', 'cauliflower', 'cabbage',
'lettuce', 'raddish', 'carrot', 'peas'])
⟩⟩⟩ price.values()
dict_values([40, 30, 20, 70, 50, 40, 30, 20, 80])
⟩⟩⟩ price.items()
dict_items([('tomato', 40), ('cucumber', 30), ('potato', 20),
('cauliflower', 70), ('cabbage', 50), ('lettuce', 40), ('raddish', 30),
('carrot', 20), ('peas',80)])
```

Dictionary

- The search in a dictionary is based on the key. Therefore, in a dictionary, the keys are required to be unique.
- As keys in a dictionary are immutable, lists cannot be used as keys.
- Keys in a dictionary may be of heterogeneous types, for example:
  ⟩⟩⟩ counting = {1:'one', 'one':1, 2:'two', 'two':2}

Dictionary Operations

- some operations that can be applied to a dictionary and illustrate these operations using a dictionary of digit-name pairs:
  digits = {0:'Zero', 1:'One', 2:'Two', 3:'Three', 4:'Four', 5:'Five', 6:'Six', 7:'Seven', 8:'Eight', 9:'Nine'}

Operation	**Examples**
Length operator **len** (number of key-value pairs in dictionary)	⟩⟩⟩ len(digits) 10
Indexing	⟩⟩⟩ digits[1] 'One'
Function **min**	⟩⟩⟩ min(digits) 0
Function **max**	⟩⟩⟩ max(digits) 9
Function **sum** (assuming keys are compatible for addition)	⟩⟩⟩ sum(digits) 45

Dictionary Operations

Operation	Examples
Membership operator **in**	⟩⟩⟩ 5 in digits True ⟩⟩⟩ 'Five' in digits False

Table 6: Summary of operations that can be applied on dictionaries

- Consider a dictionary named as winter:
  ⟩⟩⟩ winter = {11:'November ', 12: 'December', 1:'January', 2:'February'}
- Membership operation **in**, and functions **min**, **max** and **sum** apply only to the keys in a dictionary.

Dictionary Operations

- Thus, applying these operations on the dictionary winter is equivalent to applying them on winter.keys() as shown below:

```
>>> 2 in winter, min(winter), max(winter), sum(winter)
(True, 1, 12, 26)
>>> 2 in winter.keys(), min(winter.keys()), max(winter.keys()),
sum(winter.keys())
(True, 1, 12, 26)
```

- We may remove a key-value pair from a dictionary using del operator, for example:

```
>>> del winter[11]
>>> winter
{12: 'December', 1: 'January', 2: 'February'}
```

- To delete a dictionary, we use del operator:

```
>>> del winter
```

Dictionary Functions

Function	**Explanation**
D.items()	Returns an object comprising of tuples of **key-value** pairs present in dictionary D
D.keys()	Returns an object comprising of all keys of dictionary D
D.values()	Returns an object comprising of all values of dictionary D
D.clear()	Removes all **key-value** pairs from dictionary D
D.get(key, default)	For the specified **key**, the function returns the associated value. Returns the **default** value in the case **key** is not present in the dictionary D
D.copy()	Creates a shallow copy of dictionary D
D1.update(D2)	Adds the **key-value** pairs of dictionary D2 to dictionary D1

Table 7: Dictionary Functions

Inverted Dictionary

- Suppose we are maintaining a dictionary of words and their meanings of the form **word:meaning.** For simplicity, we assume that each word has a single meaning.
- There might be a few words that may have shared meaning. Given this dictionary, we wish to find synonyms of a word, i.e. given a word, we wish to find the list of words that have the same meaning.
- For this purpose, we would like to build an inverted dictionary **invDict** of **meaning:list-of-words**.

  ⟩⟩⟩Enter word meaning dictionary: {'dubious':'doubtful', 'hilarious':'amusing', 'suspicious':'doubtful', 'comical':'amusing', 'hello':'hi'}
  Inverted Dictionary:
  {'doubtful': ['dubious', 'suspicious'], 'amusing': ['hilarious', 'comical']}

Inverted Dictionary Code

```python
def buildInvDict(dict1):
    '''
    Objective: To construct inverted dictionary
    Input parameter: dict1: dictionary
    Return value: invDict: dictionary
    '''
    invDict = {}
    for key,values in dict1.items():
        if value in invDict:
            invDict[value].append(key)
        else:
            invDict[value] = [key]
    invDict={x:invDict[x] for x in invDict if len(invDict[x]>1)
    }
    return invDict
```

Inverted Dictionary Code

```
def main():
    '''
    Objective: To find inverted dictionary
    Input parameters: None
    Return value: None
    '''
    wordMeaning = eval(input('Enter word meaning dictionary:'))
    meaningWord = buildInvDict(wordMeaning)
    print('Inverted dictionary:\n', meaningWord)

# Statements to initiate the call to main function.

if __name__=='__main__':
    main()
```

References

Python Programming: A modular approach by Taneja Sheetal, and Kumar Naveen, *Pearson Education India, Inc.*, 2017.

Thank You
Any Questions?