UNIT – 4

UNIT IV LISTS, TUPLES, DICTIONARIES **9**

Lists: list operations, list slices, list methods, list loop, mutability, aliasing, cloning lists, list parameters; Tuples: tuple assignment, tuple as return value; Dictionaries: operations and methods; advanced list processing - list comprehension; Illustrative programs: selection sort, insertion sort, merge sort, histogram.

LISTS, TUPLES, DICTIONARIES

4.1 LISTS:

A list is a sequence of values. In a string, the values are characters but in a list, list values can be any type.

Elements or Items:

The values in a list are called elements or items. list must be enclosed in square brackets ([and]).

Examples:

```
>>>[10,20,30]
>>>['hi','hello','welcome']
```

Nested List:

A list within another list is called nested list.

Example:

```
['good',10,[100,99]]
```

Empty List:

A list that contains no elements is called empty list. It can be created with empty brackets, [].

Assigning List Values to Variables:

Example:

```
>>>numbers=[40,121]
>>>characters=['x','y']
>>>print(numbers,characters)
```

Output:

```
[40,12]['x','y']
```

4.1.1 LIST OPERATIONS:

Example 1: The + operator concatenates lists:

```
>>>a=[1,2,3]
>>>b=[4,5,6]
>>>c=a+b
>>>c
```

Output: [1,2,3,4,5,6]

Example 2:

The * operator repeats a list given number of times:

```
>>>[0]*4
```

Output:

```
[0,0,0,0]
```

```
>>>[1,2,3]*3
```

Output:

```
[1,2,3,1,2,3,1,2,3]
```

The first example repeats [0] four times.The second example repeats [1,2,3] three times.

4.1.2 LIST SLICES:

List slicing is a computationally fast way to methodically access parts of given data.

Syntax:

Listname[start:end:step]where **:end** represents the first value that is not in the selected slice. The differencebetween end and start is the numbe of elements selected (if step is 1, the default).The start and end may be a negative number. For negative numbers, the count starts from the end of the array instead of the beginning.

Example:

```
>>>t=[‘a’,’b’,’c’,’d’,’e’,’f’]
```

Lists are mutable, so it is useful to make a copy before performing operations that modify this. A slice operator on the left side of an assignment can update multiple elements.

Example:

```
>>>t[1:3]=[‘x’,’y’]
>>>t
```

Output:

```
[‘a’,’x’,’y’,’d’,’e’,’f’]
```

4.1.3 LIST METHODS:

Python provides methods that operate on lists.

Example:

1. append adds a new element to the end of a list.

```
>>>t=[‘a’,’b’,’c’,’d’]
>>>t.append(‘e’)
>>>t
```

Output:

```
[‘a’,’b’,’c’,’d’,’e’]
```

2. extend takes a list as an argument and appends all the elements.

```
>>>t1=[‘a’,’b’]
>>>t2=[‘c’,’d’]
>>>t1.extend(t2)
>>>t1
```

Output:

```
[‘a’,’b’,’c’,’d’]
```

t2 will be unmodified.

4.1.4 LIST LOOP:

List loop is traversing the elements of a list with a for loop.

Example 1:

```
>>>mylist=[[1,2],[4,5]]
>>>for x in mylist:
        if len(x)==2:
                print x
```

Output:

```
[1,2]
[4,5]
```

Example 2:

```
>>>for i in range(len(numbers)):
        Numbers[i]=numbers[i]*2
```

Above example is used for updating values in numbers variables.Above loop traverses the list and updates each element. Len returns number of elements in the list. Range returns a list of indices from 0 to n-1, where n is the length of the list.Each time through the loop i gets the index of next element. A for loop over an empty list never runs the body:

Example:

```
for x in []:
        print('This won't work')
```

A list inside another list counts as a single element. The length of the below list is four:

```
['spam',1,['x','y'],[1,2,3]]
```

Example 3:

```
colors=["red","green","blue","purple"]
for i in range(len(colors)):
        print(colors[i])
```

Output:

```
red
green
blue
purple
```

4.1.5 MUTABILITY:

Lists are mutable. Mutable means, we can change the content without changing the identity. Mutability is the ability for certain types of data to be changed without entirely recreating it.Using mutable data types can allow programs to operate quickly and efficiently.

Example for mutable data types are:

List, Set and Dictionary

Example 1:

```
>>>numbers=[42,123]
>>>numbers[1]=5
>>>numbers
[42,5]
```

Here, the second element which was 123 is now turned to be 5.

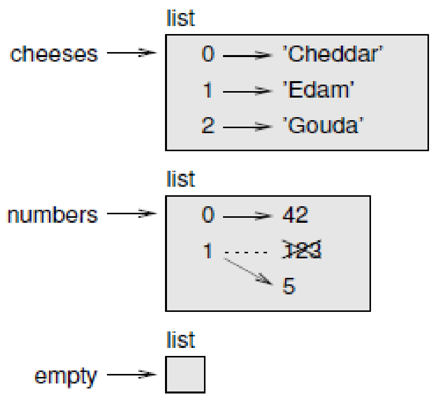


Figure 4.1 shows the state diagram for cheeses, numbers and empty:

Lists are represented by boxes with the word "list" outside and the elements of the list inside. cheeses refers to a list with three elements indexed 0, 1 and 2.

numbers contains two elements; the diagram shows that the value of the second element has been reassigned from 123 to 5. empty refers to a list with no elements.

The in operator also works on lists.

```
>>> cheeses = ['Cheddar', 'Edam', 'Gouda']
>>> 'Edam' in cheeses
True
>>> 'Brie' in cheeses
False
```

4.1.6 ALIASING

If a refers to an object and we assign b = a, then both variables refer to the same object:

```
>>> a = [1, 2, 3]
>>> b = a
>>> b is a
True
```

The state diagram looks like Figure 4.2.

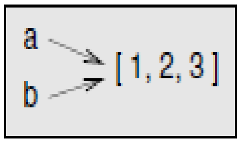


Figure 4.2 State Diagram

The association of a variable with an object is called a **reference**. In this example, there are two references to the same object.An object with more than one reference has more than one name, so we say that the object is **aliased**.If the aliased object is mutable, changes made with one alias affect the other:

4.1.7 CLONING LISTS Copy list v Clone list

veggies=["potatoes","carrots","pepper","parsnips","swedes","onion","minehead"]
veggies[1]="beetroot"

# Copying a list gives it another name

daniel=veggies

# Copying a complete slice CLONES a list

david=veggies[:]

daniel[6]="emu"

# Daniel is a SECOND NAME for veggies so that changing Daniel also changes veggies.

# David is a COPY OF THE CONTENTS of veggies so that changing Daniel (or veggies) does NOT change David.

# Changing carrots into beetroot was done before any of the copies were made, so will affect all of veggies, daniel and david

```
for display in (veggies, daniel, david):
    print(display)
```

Output:

```
['potatoes', 'beetroot', 'pepper', 'parsnips', 'swedes', 'onion', 'emu']
['potatoes', 'beetroot', 'pepper', 'parsnips', 'swedes', 'onion', 'emu']
['potatoes', 'beetroot', 'pepper', 'parsnips', 'swedes', 'onion', 'minehead']
```

4.1.8 LIST PAPRAMETERS

When we pass a list to a function, the function gets a reference to the list. If the function modifies the list, the caller sees the change. For example, delete_head removes the first element from a list:

Example:

```
def delete_head(t):
        del t[0]
```

Here's how it is used:

```
>>> letters = ['a', 'b', 'c']
>>> delete_head(letters)
>>> letters
```

Output:

```
['b', 'c']
```

The parameter t and the variable letters are aliases for the same object. The stack diagram looks like Figure 4.3.Since the list is shared by two frames, I drew it between them. It is important to distinguish between operations that modify lists and operations that create new lists. For example, the append method modifies a list, but the + operator creates a new list.

Example 1:

```
>>> t1 = [1, 2]
>>> t2 = t1.append(3)
>>> t1
```

Output:

```
[1, 2, 3]
```

Example 2:

```
>>> t2
```

Output:

```
None
```

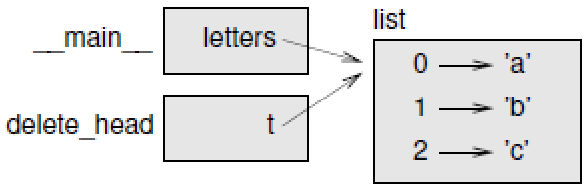


Figure 4.3 Stack Diagram

4.2. TUPLES

- A tuple is a sequence of values.
- The values can be any type and are indexed by integers, unlike lists.
- Tuples are immutable.

Syntactically, a tuple is a comma-separated list of values:

```
>>> t = 'a', 'b', 'c', 'd', 'e'
```

Although it is not necessary, it is common to enclose tuples in parentheses:

```
>>> t = ('a', 'b', 'c', 'd', 'e')
```

To create a tuple with a single element, we have to include a final comma:

```
>>> t1 = 'a',
>>> type(t1)
```

Output:

```
<class 'tuple'>
```

A value in parentheses is not a tuple:

```
>>> t2 = ('a')
>>> type(t2)
```

Output:

```
<class 'str'>
```

Another way to create a tuple is the built-in function tuple. With no argument, it creates an empty tuple.

Example:

```
>>> t = tuple()
>>> t
```

Output: ()

If the argument is a sequence (string, list or tuple), the result is a tuple with the elements of the sequence:

Example:

```
>>> t = tuple('lupins')
>>> t
```

Output: ('l', 'u', 'p', 'i', 'n', 's')

Because tuple is the name of a built-in function, we should avoid using it as a variable name. Most list operators also work on tuples. The bracket operator indexes an element:

Example:

```
>>> t = ('a', 'b', 'c', 'd', 'e')
>>> t[0]
```

Output: 'a' And the slice operator selects a range of elements.

Example:

>>> t[1:3] **Output:**('b', 'c')But if we try to modify one of the elements of the tuple, we get an error:

```
>>> t[0] = 'A'
```

TypeError: object doesn't support item assignmentBecause tuples are immutable, we can’t modify the elements. But we can replace one
tuple with another: **Example:**

```
>>> t = ('A',) + t[1:]
>>> t
```

Output:

('A', 'b', 'c', 'd', 'e')This statement makes a new tuple and then makes t refer to it.

The relational operators work with tuples and other sequences; Python starts by comparing the first element from each sequence. If they are equal, it goes on to the next elements, and so on, until it finds elements that differ. Subsequent elements are not considered (even if they are really big).

Example 1:

```
>>> (0, 1, 2) < (0, 3, 4)
```

Output:

```
True
```

Example 2:

```
>>> (0, 1, 2000000) < (0, 3, 4)
```

Output:

```
True
```

4.2.1 TUPLE ASSIGNMENT

It is often useful to swap the values of two variables. With conventional assignments, we have to use a temporary variable. For example, to swap a and b:

Example:

```
>>> temp = a
>>> a = b
>>> b = temp
```

This solution is cumbersome; **tuple assignment** is more elegant:

```
>>> a, b = b, a
```

The left side is a tuple of variables; the right side is a tuple of expressions. Each value is assigned to its respective variable. All the expressions on the right side are evaluated before any of the assignments.The number of variables on the left and the number of values on the right have to be the same:

```
>>> a, b = 1, 2, 3
```

ValueError: too many values to unpack

More generally, the right side can be any kind of sequence (string, list or tuple). For example, to split an email address into a user name and a domain, we could write:

```
>>> addr = 'monty@python.org'
>>> uname, domain = addr.split('@')
```

The return value from split is a list with two elements; the first element is assigned to uname, the second to domain.

```
>>> uname
'monty'
>>> domain
'python.org'
```

4.2.2 TUPLE AS RETURN VALUES

A function can only return one value, but if the value is a tuple, the effect is the same as returning multiple values. For example, if we want to divide two integers and compute the quotient and remainder, it is inefficient to compute x/y and then x%y. It is better to compute them both at the same time. The built-in function divmod takes two arguments and returns a tuple of two values, the quotient and remainder. We can store the result as a tuple:

Example:

```
>>> t = divmod(7, 3)
>>> t
```

Ouput:

```
(2, 1)
```

Or use tuple assignment to store the elements separately:

Example:

```
>>> quot, rem = divmod(7, 3)
>>> quot
```

Output:

```
2
```

Example:

```
>>> rem
```

Output:

```
1
```

Here is an example of a function that returns a tuple:

```
def min_max(t):
    return min(t), max(t)
```

max and min are built-in functions that find the largest and smallest elements of a sequence. min_max computes both and returns a tuple of two values.

4.3 DICTIONARIES

Dictionaries have a method called items that returns a sequence of tuples, where each tuple is a key-value pair.

4.3.1 OPERATIONS AND METHODS

Example:

```
>>> d = {'a':0, 'b':1, 'c':2}
>>> t = d.items()
>>> t
```

Output:

```
dict_items([('c', 2), ('a', 0), ('b', 1)])
```

The result is a dict_items object, which is an iterator that iterates the key-value pairs. We can use it in a for loop like this:

Example:

```
>>> for key, value in d.items():
... print(key, value)
```

Output:

```
c 2
a 0
b 1
```

As we should expect from a dictionary, the items are in no particular order.

Going in the other direction, we can use a list of tuples to initialize a new dictionary:

Example:

```
>>> t = [('a', 0), ('c', 2), ('b', 1)]
>>> d = dict(t)
>>> d
```

Output:

```
{'a': 0, 'c': 2, 'b': 1}
```

Combining dict with zip yields a concise way to create a dictionary:

Example:

```
>>> d = dict(zip('abc', range(3)))
>>> d
```

Output:

```
{'a': 0, 'c': 2, 'b': 1}
```

The dictionary method update also takes a list of tuples and adds them, as key-value pairs, to an existing dictionary. It is common to use tuples as keys in dictionaries (primarily because we can’t use lists). **For example**, a telephone directory might map from last-name, first-name pairs to telephone numbers. Assuming that we have defined last, first and number, we could write: directory [last, first] = number

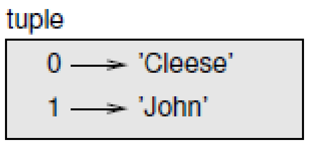
The expression in brackets is a tuple. We could use tuple assignment to traverse this dictionary.

```
For last, first in directory:
print(first, last, directory[last,first])
```

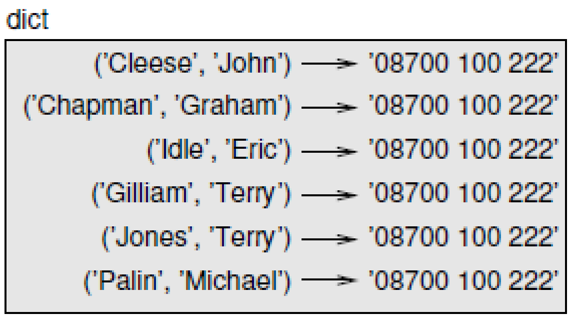
This loop traverses the keys in directory, which are tuples. It assigns the elements of each tuple to last and first, then prints the name and corresponding telephone number.

There are two ways to represent tuples in a state diagram. The more detailed version shows the indices and elements just as they appear in a list. For example, the tuple('Cleese', 'John') would appear as in Figure 4.4.But in a larger diagram we might want to leave out the details. For example, a diagram of the telephone directory might appear as in Figure 4.5.

Here the tuples are shown using Python syntax as a graphical shorthand. The telephone number in the diagram is the complaints line for the BBC, so please don’t call it.



4.4 State Diagram


dict
('Cleese', 'John') → '08700 100 222'
('Chapman', 'Graham') → '08700 100 222'
('Idle', 'Eric') → '08700 100 222'
('Gilliam', 'Terry') → '08700 100 222'
('Jones', 'Terry') → '08700 100 222'
('Palin', 'Michael') → '08700 100 222'

4.5 State Diagram

4.4 ADVANCED LIST PROCESSING

List processing is a list of programming codes, including abstract data structure, used to calculate specified variables in a certain order. A value may repeat more than once.

4.4.1 LIST COMPREHENSION

The function shown below takes a list of strings, maps the string method capitalize to the elements, and returns a new list of strings:

```
def capitalize_all(t):
    res = []
    for s in t:
        res.append(s.capitalize())
    return res
```

We can write this more concisely using a **list comprehension**:

```
def capitalize_all(t):
    return [s.capitalize() for s in t]
```

The bracket operators indicate that we are constructing a new list. The expression inside the brackets specifies the elements of the list, and the ‘for’ clause indicates what sequence we are traversing.The syntax of a list comprehension is a little awkward because the loop variable, s in this example, appears in the expression before we get to the definition.

List comprehensions can also be used for filtering. For example, this function selects only the elements of t that are upper case, and returns a new list:

```
def only_upper(t):
    res = []
    for s in t:
        if s.isupper():
            res.append(s)
    return res
```

We can rewrite it using a list comprehension

```
def only_upper(t):
    return [s for s in t if s.isupper()]
```

List comprehensions are concise and easy to read, at least for simple expressions. And they are usually faster than the equivalent for loops, sometimes much faster.

But, list comprehensions are harder to debug because we can’t put a print statement inside the loop. We can use them only if the computation is simple enough that we are likely to get it right the first time.

4.5 ILLUSTRATIVE PROGRAMS   SELECTION SORT

Selection sort is one of the simplest sorting algorithms. It is similar to the hand picking where we take the smallest element and put it in the first position and the second smallest at the second position and so on. It is also similar. We first check for smallest element in the list and swap it with the first element of the list. Again, we check for the smallest number in a sub list, excluding the first element of the list as it is where it should be (at the first position) and put it in the second position of the list. We continue repeating this process until the list gets sorted.

ALGORITHM:

1. Start from the first element in the list and search for the smallest element in the list.
2. Swap the first element with the smallest element of the list.
3. Take a sub list (excluding the first element of the list as it is at its place) and search for the smallest number in the sub list (second smallest number of the entire list) and swap it with the first element of the list (second element of the entire list).
4. Repeat the steps 2 and 3 with new subsets until the list gets sorted.

PROGRAM:

```
a = [16, 19, 11, 15, 10, 12, 14]
i = 0
while i<len(a):
   #smallest element in the sublist
   smallest = min(a[i:])
   #index of smallest element
   index_of_smallest = a.index(smallest)
   #swapping
   a[i],a[index_of_smallest] = a[index_of_smallest],a[i]
   i=i+1
print (a)
```

Output

```
>>>
 [10, 11, 12, 14, 15, 16, 19]
```

4.5.2 INSERTION SORT

Insertion sort is similar to arranging the documents of a bunch of students in order of their ascending roll number. Starting from the second element, we compare it with the first element and swap it if it is not in order. Similarly, we take the third element in the next iteration and place it at the right place in the sub list of the first and second elements (as the sub list containing the first and second elements is already sorted). We repeat this step with the fourth element of the list in the next iteration and place it at the right position in the sub list containing the first, second and the third elements. We repeat this process until our list gets sorted.

ALGORITHM:

1. Compare the current element in the iteration (say A) with the previous adjacent element to it. If it is in order then continue the iteration else, go to step 2.
2. Swap the two elements (the current element in the iteration (A) and the previous adjacent element to it).
3. Compare A with its new previous adjacent element. If they are not in order, then proceed to step 4.
4. Swap if they are not in order and repeat steps 3 and 4.
5. Continue the iteration.

PROGRAM:

```
a = [16, 19, 11, 15, 10, 12, 14]
#iterating over a
for i in a:
   j = a.index(i)
   #i is not the first element
   while j>0:
      #not in order
      if a[j-1] > a[j]:
            #swap
            a[j-1],a[j] = a[j],a[j-1]
        else:
            #in order
            break
        j = j-1
print (a)
```

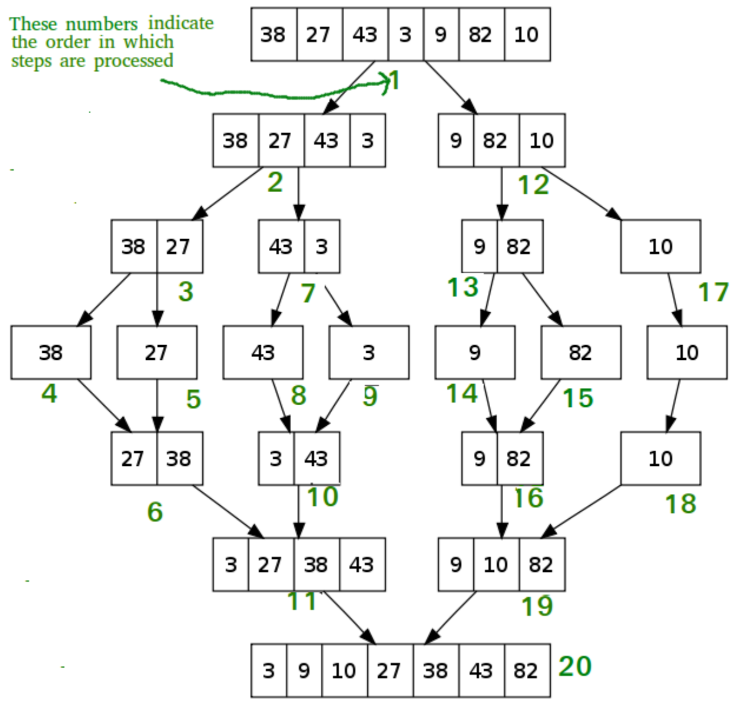
Output

```
 >>>
[10, 11, 12, 14, 15, 16, 19]
```

4.5.3 MERGE SORT

Merge Sort is a Divide and Conquer algorithm. It divides input array in two halves, calls itself for the two halves and then merges the two sorted halves. **The merge() function** is used for merging two halves. The merge(arr, l, m, r) is key process that assumes that arr[l..m] and arr[m+1..r] are sorted and merges the two sorted sub-arrays into one. See following C implementation for details.

The following diagram shows the complete merge sort process for an example array {38, 27, 43, 3, 9, 82, 10}. If we take a closer look at the diagram, we can see that the array is recursively divided in two halves till the size becomes 1. Once the size becomes 1, the merge processes comes into action and starts merging arrays back till the complete array is merged.



ALGORITHM:

```
MergeSort(arr[], l,  r)
If r > l
     1. Find the middle point to divide the array into two halves:
             middle m = (l+r)/2
     2. Call mergeSort for first half:
             Call mergeSort(arr, l, m)
     3. Call mergeSort for second half:
             Call mergeSort(arr, m+1, r)
     4. Merge the two halves sorted in step 2 and 3:
             Call merge(arr, l, m, r)
```

PROGRAM:

```
def mergeSort(alist):
   print("Splitting ",alist)
   if len(alist)>1:
      mid = len(alist)//2
      lefthalf = alist[:mid]
      righthalf = alist[mid:]
      mergeSort(lefthalf)
      mergeSort(righthalf)
      i=0
      j=0
      k=0
      while i < len(lefthalf) and j < len(righthalf):
         if lefthalf[i] < righthalf[j]:
            alist[k]=lefthalf[i]
            i=i+1
         else:
            alist[k]=righthalf[j]
            j=j+1
         k=k+1
      while i < len(lefthalf):
         alist[k]=lefthalf[i]
         i=i+1
         k=k+1
      while j < len(righthalf):
         alist[k]=righthalf[j]
         j=j+1
         k=k+1
   print("Merging ",alist)
n = input("Enter the size of the list: ")
n=int(n);
alist = []
for i in range(n):
 alist.append(input("Enter %dth element: "%i))
mergeSort(alist)
print(alist)
```

Input:

```
a = [16, 19, 11, 15, 10, 12, 14]
```

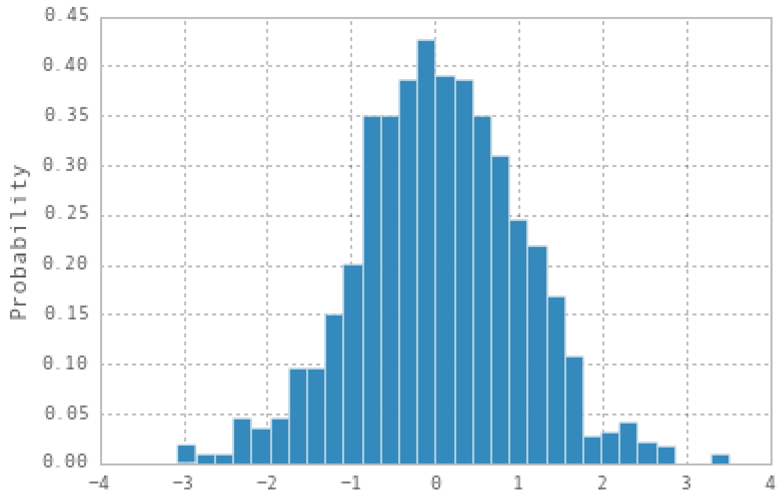
Output:

```
 >>>
[10, 11, 12, 14, 15, 16, 19]
```

4.5.5 HISTOGRAM

- A histogram is a visual representation of the Distribution of a Quantitative variable.
    - Appearance is similar to a vertical bar graph, but used mainly for continuous distribution
    - It approximates the distribution of variable being studied
- A visual representation that gives a discretized display of value counts.

Example:



```
def histogram(s):
        d = dict()
        for c in s:
                if c not in d:
                        d[c] = 1
                else:
                        d[c] += 1
                return d
```

The name of the function is histogram, which is a statistical term for a collection of counters (or frequencies).

The first line of the function creates an empty dictionary. The for loop traverses the string.Each time through the loop, if the character c is not in the dictionary, we create a new item with key c and the initial value 1 (since we have seen this letter once). If c is already in the dictionary we increment d[c]. Here’s how it works:

Example:

```
>>> h = histogram('brontosaurus')
>>> h
```

Output:

```
{'a': 1, 'b': 1, 'o': 2, 'n': 1, 's': 2, 'r': 2, 'u': 2, 't': 1}
```

The histogram indicates that the letters 'a' and 'b' appear once; 'o' appears twice, and so on. Dictionaries have a method called get that takes a key and a default value. If the key appears in the dictionary, get returns the corresponding value; otherwise it returns the default value. **For example:**

```
>>> h = histogram('a')
>>> h
{'a': 1}
>>> h.get('a', 0)
1
>>> h.get('b', 0)
0
```

As an exercise, use get to write histogram more concisely. We should be able to eliminate the if statement.

If we use a dictionary in a for statement, it traverses the keys of the dictionary. For example, print_hist prints each key and the corresponding value:

```
def print_hist(h):
    for c in h:
        print(c, h[c])
```

Here's what the output looks like:

```
>>> h = histogram('parrot')
>>> print_hist(h)
a 1
p 1
r 2
t 1
o 1
```

Again, the keys are in no particular order. To traverse the keys in sorted order, we can use the built-in function sorted:

```
>>> for key in sorted(h):
    ... print(key, h[key])
a 1
o 1
p 1
r 2
t 1
```

Write a function named choose_from_hist that takes a histogram as defined in Histogram given above and returns a random value from the histogram, chosen with probability in proportion to frequency. For example, for this histogram:

```
>>> t = ['a', 'a', 'b']
>>> hist = histogram(t)
>>> hist
{'a': 2, 'b': 1}
```

The function should return 'a' with probability 2/3 and 'b' with probability 1/3.

TWO MARKS

1. What are elements in a list? Give example.

    The values in a list are called elements or items.
    A list must be enclosed in square brackets ([and]).
    Examples:
    ```
    >>>[10,20,30]
    >>>['hi','hello','welcome']
    ```
2. What is a nested list? Give example.

    A list within another list is called nested list.
    Example:
    ['good',10,[100,99]]
3. What is a empty list?

    A list that contains no elements is called empty list. It can be created with empty brackets, [].

4. What is list slicing? Write its syntax.

    List slicing is a computationally fast way to methodically access parts of given data. **Syntax:**
    Listname [start:end:step]
5. What is mutability? What is its use?

Mutability is the ability for certain types of data to be changed without entirely recreating it. Using mutable data types can allow programs to operate quickly and efficiently.

6. List any three mutable data types. Give example for mutability.

    Example for mutable data types are:

    List, Set and Dictionary

    Example 1:

```
>>>numbers=[42,123]
>>>numbers[1]=5
>>>numbers
```

    Output: [42,5]

7. What is aliasing? Give example.

    An object with more than one reference has more than one name, so we say that the object is **aliased**.

    If the aliased object is mutable, changes made with one alias affect the other.

    Example:

    If a refers to an object and we assign b = a, then both variables refer to the same object:

```
>>> a = [1, 2, 3]
>>> b = a
>>> b is a     True
```

8. What is the difference between copying and cloning lists?

    Copying a list gives it another name but copying a complete slice clones a list.

9. Give example for copying and cloning lists.

    Example:

```
veggies=["potatoes","carrots","pepper","parsnips","swedes","onion","minehead"]
veggies[1]="beetroot"
Copying a list
        daniel=veggies
CLONES a list
        david=veggies[:]
daniel[6]="emu"
```

10. Define Dictionaries. Give example.

    A **dictionary** is an associative array (also known as hashes). Any key of the **dictionary** is associated (or mapped) to a value. The values of a **dictionary** can be any **Python** data type.

    Example:

```
>>> d = {'a':0, 'b':1, 'c':2}
>>> t = d.items()
>>> t
```

    Output:

```
dict_items([('c', 2), ('a', 0), ('b', 1)])
```

11. What is list processing?

List processing is a list of programming codes, including abstract data structure, used to calculate specified variables in a certain order. A value may repeat more than once.