Python Programming

Chapter 3 :

Structured Types, Mutability and Higher-Order Functions

Dr Vivaksha Jariwala

Introduction

- Numeric types int and float are scalar types.
- Str can be thought of as a structured, or non-scalar type.
- We can use indexing to extract individual characters from a string
- Slicing to extract substrings.

Introduction…

We introduce four additional structured types.

1. Tuple – simple generalization of str.
2. List
3. Range
4. Dict

Tuples

- Like strings, **tuples** are ordered sequences of elements.
- The difference is that the elements of a tuple need not be characters.
- The individual elements can be of any type, and need not be of the same type as each other.

Tuples…

- Literals of type tuple are written by enclosing a comma-separated list of elements within parentheses.
- For example, we can write

```
t1 = ()
t2 = (1, 'two', 3)
print t1
print t2
```

- Unsurprisingly, the print statements produce the output

```
()
(1, 'two', 3)
```

Tuples…

- To denote the singleton tuple containing this value, we write (1,)
- Repetition can be used on tuples. 3*(‘a’,2) evaluates to (‘a’, 2, ‘a’, 2, ‘a’, 2)
- Like strings, tuples can be concatenated, indexed, and sliced. Consider

```
t1 = (1, 'two', 3)
t2 = (t1, 3.25)
print t2                 → ((1, 'two', 3), 3.25)
print (t1 + t2)          → (1, 'two', 3, (1, 'two', 3), 3.25)
print (t1 + t2)[3]       → (1, 'two', 3)
print (t1 + t2)[2:5]     → (3, (1, 'two', 3), 3.25)
```

Tuples…

- A for statement can be used to iterate over the elements of a tuple

```
Def intersect(t1,t2):
        “””Assumes t1 and t2 are tuples returns a tuple
        containing elements that are in both t1 and t2”””
        result = ( )
        for e in t1
                if e in t2
                        result += (e, )
        return result
```

Tuples…

Following code prints the common divisors of 20 and 100 and then the sum of all the divisors.

```
def findDivisors (n1, n2):
        """Assumes that n1 and n2 are positive ints
        Returns a tuple containing all common divisors of n1 &
        n2"""
        divisors = () #the empty tuple
        for i in range(1, min (n1, n2) + 1):
                if n1%i == 0 and n2%i == 0:
                        divisors = divisors + (i,)
        return divisors
divisors = findDivisors(20, 100)
print divisors
total = 0
for d in divisors:
        total += d
print total
```

Sequences and Multiple Assignment

- after executing the statement x, y = (3, 4), x will be bound to 3 and y to 4. Similarly, the statement a, b, c = 'xyz' will bind a to 'x', b to 'y', and c to 'z'.

Sequences and Multiple Assignment

```
def findExtremeDivisors(n1, n2):
      """Assumes that n1 and n2 are positive ints
      Returns a tuple containing the smallest common
      divisor > 1 and the largest common divisor of
      n1 and n2"""
      divisors = () #the empty tuple
      minVal, maxVal = None, None
      for i in range(2, min(n1, n2) + 1):
            if n1%i == 0 and n2%i == 0:
                  if minVal == None or i < minVal:
                        minVal = i
                  if maxVal == None or i > maxVal:
                        maxVal = i
      return (minVal, maxVal)
minDivisor, maxDivisor = findExtremeDivisors(100, 200)
```

Ranges

- Like strings and tuples, ranges are immutable.
- The range function returns an object of type range
- Range function takes three integer arguments : start, stop and step
- Returns the progression of integers start, start+step, start+2*step…
- If step is positive, the last element is the largest integer start + i*step less than stop.
- If step is negative, the last element is the smallest integer start + i*step greater than stop

Ranges…

- If only two arguments are supplied, a step of 1 is used.
- If only one argument is supplied, that argument is the stop, start defaults to 0 and step defaults to 1.
- All the operations on tuples are available for ranges, except for concatenation and repetition
- range(10)[2:6][2] evaluates to 4
- The most common use of range is in for loops

Lists and Mutability

- Like a tuple, a **list** is an ordered sequence of values, where each value is identified by an index.
- The syntax for expressing literals of type list is similar to that used for tuples; the difference is that we use square brackets rather than parentheses.
- The empty list is written as [], and singleton lists are written without that (oh so easy to forget) comma before the closing bracket.

Lists and Mutability…

```
L = ['I did it all', 4, 'like']
for i in range(len(L)):
        print L[i]
```

produces the output,

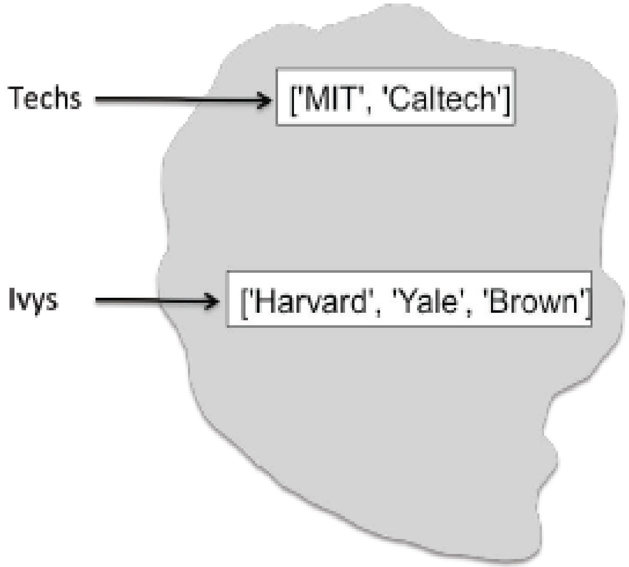
```
I did it all
4
like
```

Lists and Mutability…

- Lists differ from tuples in one hugely important way: lists are **mutable**.
- In contrast, tuples and strings are **immutable**.
- There are many operators that can be used to create objects of these immutable types, and variables can be bound to objects of these types.
- But objects of immutable types cannot be modified.
- On the other hand, objects of type list can be modified after they are created.
- [1,2,3,4][1:3][1]
- Ans???

Lists and Mutability…

- Techs = ['MIT', 'Caltech']
- Ivys = ['Harvard', 'Yale', 'Brown']



Lists and Mutability…

- The assignment statements
- Univs = [Techs, Ivys]
- Univs1 = [['MIT', 'Caltech'], ['Harvard', 'Yale', 'Brown']]
- also create new lists and bind variables to them.
- The elements of these lists are themselves lists. The three print statements

```
print('Univs =', Univs)
print('Univs1 =', Univs1)
print(Univs == Univs1)
```

- produce the output

```
Univs = [['MIT', 'Caltech'], ['Harvard', 'Yale', 'Brown']]
Univs1 = [['MIT', 'Caltech'], ['Harvard', 'Yale', 'Brown']]
True
```

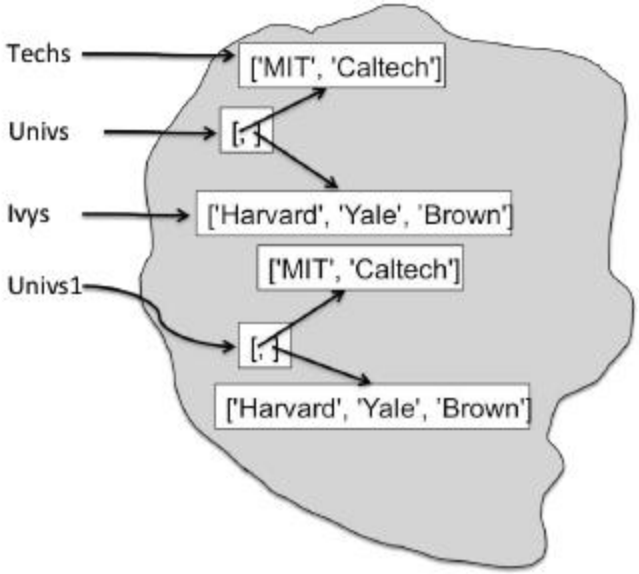
- It appears as if Univs and Univs1 are bound to the same

Lists and Mutability…

```
print (Univs == Univs1) #test value equality
print (id(Univs) == id(Univs1)) #test object equality
print ('Id of Univs =', id(Univs))
print ('Id of Univs1 =', id(Univs1))
```

it prints

```
        True
        False
Id of Univs = 24499264
Id of Univs1 = 24500504
```



Lists and Mutability…

```
L1 = [1,2,3]
L2 = [4,5,6]
L3 = L1 + L2
print ('L3 =', L3)
L1.extend(L2)
print ('L1 =', L1)
L1.append(L2)
print ('L1 =', L1)
```

will print

```
L3 = [1, 2, 3, 4, 5, 6]
L1 = [1, 2, 3, 4, 5, 6]
L1 = [1, 2, 3, 4, 5, 6, [4, 5, 6]]
```

Lists and Mutability…

- **L.append(e)** adds the object e to the end of L.
- **L.count(e)** returns the number of times that e occurs in L.
- **L.insert(i, e)** inserts the object e into L at index i.
- **L.extend(L1)** adds the items in list L1 to the end of L.
- **L.remove(e)** deletes the first occurrence of e from L.
- **L.index(e)** returns the index of the first occurrence of e in L. It raises an exception if e is not in L.
- **L.pop(i)** removes and returns the item at index i in L. If i is omitted, it defaults to -1, to remove and return the last element of L.
- **L.sort()** sorts the elements of L in ascending order.
- **L.reverse()** reverses the order of the elements in L.

Cloning

```
def removeDups(L1, L2):
        """Assumes that L1 and L2 are lists.
        Removes any element from L1 that also occurs in L2"""
        for e1 in L1:
                if e1 in L2:
                        L1.remove(e1)
L1 = [1,2,3,4]
L2 = [1,2,5,6]
removeDups(L1, L2)
print ('L1 =', L1)
```

You might be surprised to discover that the print statement produces the output

```
L1 = [2, 3, 4]
```

Cloning…

- During a for loop,
- the implementation of Python keeps track of where it is in the list using an internal counter that is incremented at the end of each iteration.
- When the value of the counter reaches the current length of the list, the loop terminates.

Cloning…

- The hidden counter starts out at 0, discovers that L1[0] is in L2, and removes it—reducing the length of L1 to 3.
- The counter is then incremented to 1, and the code proceeds to check if the value of L1[1] is in L2.
- Notice that this is not the original value of L1[1] (i.e., 2), but rather the current value of L1[1] (i.e., 3).

Cloning…

- One way to avoid this kind of problem is to use slicing to **clone** (i.e., make a copy of) the list and write for e1 in L1[:].
- Notice that writing

```
newL1 = L1
for e1 in newL1
```

- would not have solved the problem.
- It would not have created a copy of L1, but would merely have introduced a new name for the existing list.
- Slicing is not the only way to clone lists in Python.

List Comprehension

- Provides a concise way to apply an operation to the values in a sequence.
- It creates a new list in which each element is the result of applying a given operation to a value from a sequence (e.g., the elements in another list).
- For example,
- L = [x**2 for x in range(1,7)]
- print L
- will print the list
- [1, 4, 9, 16, 25, 36]

List Comprehension...

- For example, the code

```
mixed = [1, 2, 'a', 3, 4.0]
print [x**2 for x in mixed if type(x) == int]
```

- squares the integers in mixed, and then prints [1, 4, 9].

Example of List

- A python program to display the elements of a list in reverse order.

Example…

```
days = [‘Sunday’, ‘Monday’, ‘Tuesday’, ‘Wednesday’, ‘Thursday’]
print(‘\n In reverse order: ‘)
i=len(days)-1
while i>=0:
        print(days[i])
        i-=1
print(‘\n In Reverse order : ‘)
i=-1
while i>=-len(days):
        print(days[i])
        i-=1
```

Exercise

- A python program to find maximum and minimum elements in a list of elements.

Example

- A python program to create a list with employee data and then retrieve a particular employee details.

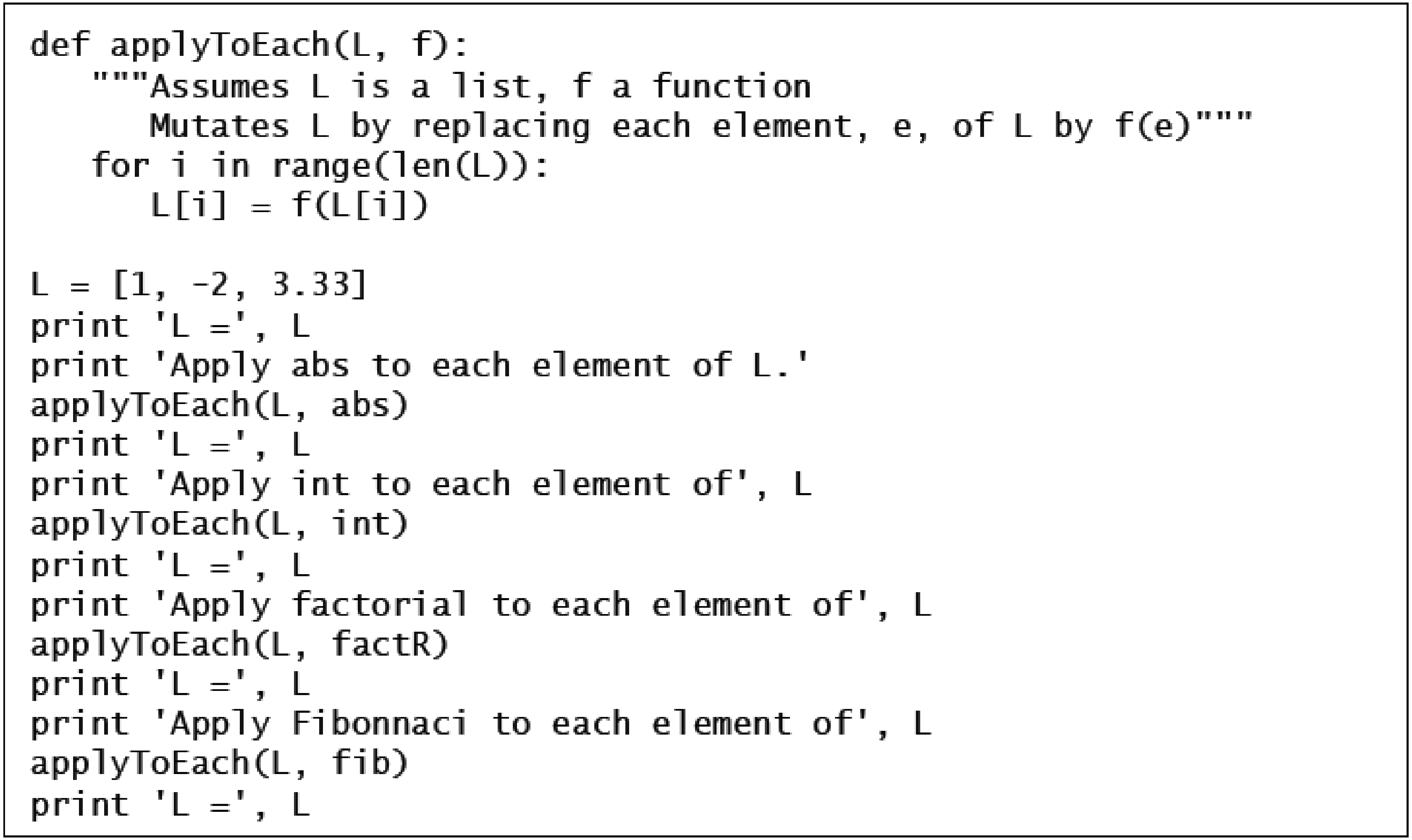
Example…

```
emp= [ ]
n=int(input(‘How many employees” ‘))
for i in range(n):
        print(‘Enter ID : ‘, end=‘ ’)
        emp.append(int(input()))
        print(‘Enter name : ‘, end=‘ ’)
        emp.append(int(input()))
        print(‘Enter salary : ‘, end=‘ ’)
        emp.append(int(input()))
print(‘The list is created with employee data.’)
id=int(input(Enter employee id: ‘))
for i in range(len(emp)):
        if id==emp[i]:
                print(‘Id={:d},Name={:s},salary={:d}’.format(emp[i],emp[i+1],emp[i+2]))
                break
```

Functions as Objects

- In Python, functions are **first-class objects**.
- That means that they can be treated like objects of any other type, e.g., int or list.
- It allows a style of coding called **higher-order programming**.

Functions as Objects...



```
def applyToEach(L, f):
    """Assumes L is a list, f a function
       Mutates L by replacing each element, e, of L by f(e)"""
    for i in range(len(L)):
        L[i] = f(L[i])

L = [1, -2, 3.33]
print 'L =', L
print 'Apply abs to each element of L.'
applyToEach(L, abs)
print 'L =', L
print 'Apply int to each element of', L
applyToEach(L, int)
print 'L =', L
print 'Apply factorial to each element of', L
applyToEach(L, factR)
print 'L =', L
print 'Apply Fibonnaci to each element of', L
applyToEach(L, fib)
print 'L =', L
```

Functions as Objects...

- L = [1, -2, 3.3300000000000001]
- Apply abs to each element of L.
- L = [1, 2, 3.3300000000000001]
- Apply int to each element of [1, 2, 3.3300000000000001]
- L = [1, 2, 3]
- Apply factorial to each element of [1, 2, 3]
- L = [1, 2, 6]
- Apply Fibonnaci to each element of [1, 2, 6]
- L = [1, 2, 13]

Functions as Objects…

- Python has a built-in higher-order function, map, that is similar to, but more general than, the applyToEach function defined
- In its simplest form the first argument to map is a unary function (i.e., a function that has only one parameter) and the second argument is any ordered collection of values suitable as arguments to the first argument.
- It returns a list generated by applying the first argument to each element of the second argument.

Functions as Objects…

```
L1=[1, 28, 36]
L2=[2, 57, 9]
for i in map(min, L1, L2):
    print(i)
```

Prints

```
1
28
9
```

Functions as Objects…

- Python supports the creation of anonymous functions. (i.e. functions that are not bound to a name)
- That can be done using reserved word lambda.
- General form is

  Lambda <sequence of variable names>: <expression>

Functions as Objects...

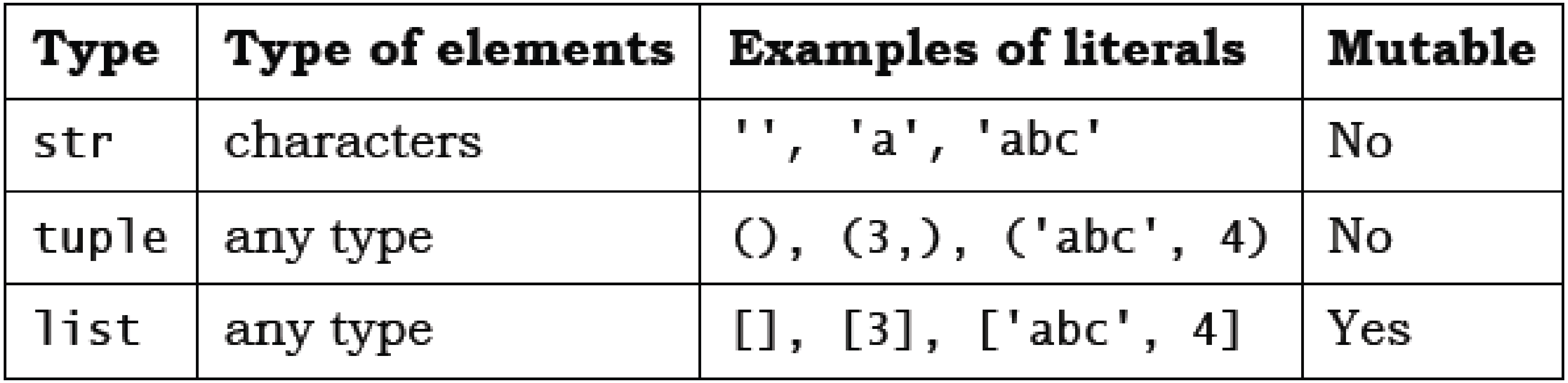
```
L=[ ]
for i in map(lambda x, y: x**y, [1, 2, 3, 4], [3, 2, 1, 0]):
    L.append(i)
print(L)
```

- Prints [1, 4, 3, 1]

Strings, Tuples, Ranges and Lists

- **seq[i]** returns the ith element in the sequence.
- **len(seq)** returns the length of the sequence.
- **seq1 + seq2** returns the concatenation of the two sequences.
- **n * seq** returns a sequence that repeats seq n times.
- **seq[start:end]** returns a slice of the sequence.
- **e in seq** is True if e is contained in the sequence and False otherwise.
- **e not in seq** is True if e is not in the sequence and False otherwise.
- **for e in seq** iterates over the elements of the

Strings, Tuples, Ranges and Lists…



Type	Type of elements	Examples of literals	Mutable
str	characters	'', 'a', 'abc'	No
tuple	any type	(), (3,), ('abc', 4)	No
list	any type	[], [3], ['abc', 4]	Yes

Strings, Tuples, Ranges and Lists…

- Python programmers tend to use lists far more often than tuples.
- Since lists are mutable, they can be constructed incrementally during a computation.
- For example, the following code incrementally builds a list containing all of the even numbers in another list.

```
evenElems = []
for e in L:
        if e%2 == 0:
                evenElems.append(e)
```

Strings, Tuples, Ranges and Lists…

- One advantage of tuples is that because they are immutable, aliasing is never a worry.
- Another advantage of their being immutable is that tuples, unlike lists, can be used as keys in dictionaries, as we will see in the next section.
- Since strings can contain only characters, they are considerably less versatile than tuples or lists.
- On the other hand, when you are working with a string of characters there are many built-in methods that make life easy.

Strings, Tuples, Ranges and Lists…

Print(‘My favorite professor - - John G. - - rocks’.split(‘ ‘))

Print(‘My favorite professor - - John G. - - rocks’.split(‘ –’))

Print(‘My favorite professor - - John G. - - rocks’.split(‘ - -‘))

- Prints

[‘My’, ‘favorite’, ‘professor - - John’, ‘G. - - rocks’]

[‘My favorite professor ‘, ‘ ‘ , ‘John G.’, ‘ ‘ , ‘rocks’]

[‘My favorite professor ‘, ‘John G.’, ‘rocks’]

Strings, Tuples, Ranges and Lists…

- **s.count(s1)** counts how many times the string s1 occurs in s.
- **s.find(s1)** returns the index of the first occurrence of the substring s1 in s, and -1 if s1 is not in s.
- **s.rfind(s1)** same as find, but starts from the end of s (the “r” in rfind stands for reverse).
- **s.index(s1)** same as find, but raises an exception if s1 is not in s.
- **s.rindex(s1)** same as index, but starts from the end of s.
- **s.lower()** converts all uppercase letters in s to lowercase.
- **s.replace(old, new)** replaces all occurrences of the string old in s with the string new.
- **s.rstrip()** removes trailing white space from s.
- **s.split(d)** Splits s using d as a delimiter. Returns a list of substrings of s. For example, the value of 'David Guttag plays basketball'.split(' ') is ['David', 'Guttag', 'plays', 'basketball']. If d is omitted, the substrings are separated by arbitrary strings of whitespace characters (space, tab, newline, return, and formfeed).

Program

- A Python program to access each element of a string in forward and reverse orders using while loop.

Program…

```
str='Core Python'
n=len(str)
i=0
while i<n:
        print(str[i], end=' ')
        i+=1
print()
#access in reverse order
i=-1
while i>=-n:
        print(str[i], end=' ')
        i-=1
print()
#access in reverse order using negative index
i=1
n=len(str)
while i<=n:
        print(str[-i], end=' ')
        i+=1
```

Output:

```
%run -i "D:/Vivaksha/Subjects/Python/ExamplePrograms/string.py"
C o r e   P y t h o n
n o h t y P   e r o C
n o h t y P   e r o C
```

Accessing elements of string using for loop

```
str='Core Python'
for i in str:
        print(i, end= ' ')
print()
for i in str[::-1]:
        print(i,end=' ')
```

To find sub string in main string

```
str=input('Enter main string: ')
sub=input('Enter sub string: ')
if sub in str:
        print(sub+ ' is found in main string')
else:
        print(sub+ ' is not found in main string')
```

```
%run -i "D:/Vivaksha/Subjects/Python/ExamplePrograms/string2.py"
Enter main string: Vivaksha Jariwala
Enter sub string: va
va is found in main string
```

Sort a group of strings into alphabetical order

```
str=[]
n=int(input('How many strings ?’))
for i in range(n):
    print('Enter string: ', end=' ')
    str.append(input())
str.sort()
str1=sorted(str)
print('Sorted list ')
for i in str1:
    print(i)
```

```
How many strings ?5
Enter string:
vivaksha
Enter string:
tattva
Enter string:
jayesh
Enter string:
bhumika
Enter string:
vishruti
Sorted list
bhumika
jayesh
tattva
vishruti
vivaksha
```

Dictionaries

- Objects of type **dict** (short for dictionary) are like lists except that “indices” need not be integers—they can be values of any immutable type.
- Since they are not ordered, we call them **keys** rather than indices.
- Think of a dictionary as a set of key/value pairs. Literals of type dict are enclosed in curly braces, and each element is written as a key followed by a colon followed by a value.

Dictionaries…

- For example, the code,

```
monthNumbers = {'Jan':1, 'Feb':2, 'Mar':3, 'Apr':4,
'May':5, 1:'Jan', 2:'Feb', 3:'Mar', 4:'Apr', 5:'May'}
print ('The third month is ' + monthNumbers[3])
dist = monthNumbers['Apr'] - monthNumbers['Jan']
print ('Apr and Jan are', dist, 'months apart‘)
```

- will print

```
The third month is Mar
Apr and Jan are 3 months apart
```

Dictionaries…

- The entries in a dict are unordered and cannot be accessed with an index.
- That’s why monthNumbers[1] unambiguously refers to the entry with the key 1 rather than the second entry.
- Like lists, dictionaries are mutable. We can add an entry by writing

  monthNumbers[‘June’]=6

  Or change entry by writing

  monthNumbers[‘May’]=‘V’

Dictionaries…

- Dictionaries are one of the great things about Python.
- They greatly reduce the difficulty of writing a variety of programs.

Dictionaries…

- Like lists, dictionaries are mutable.
- So, one must be careful about side effects.
- For example,

```
FtoE['bois'] = 'wood'
print translate('Je bois du vin rouge.', dicts, 'French to English')
```

- will print

```
I wood of wine red.
```

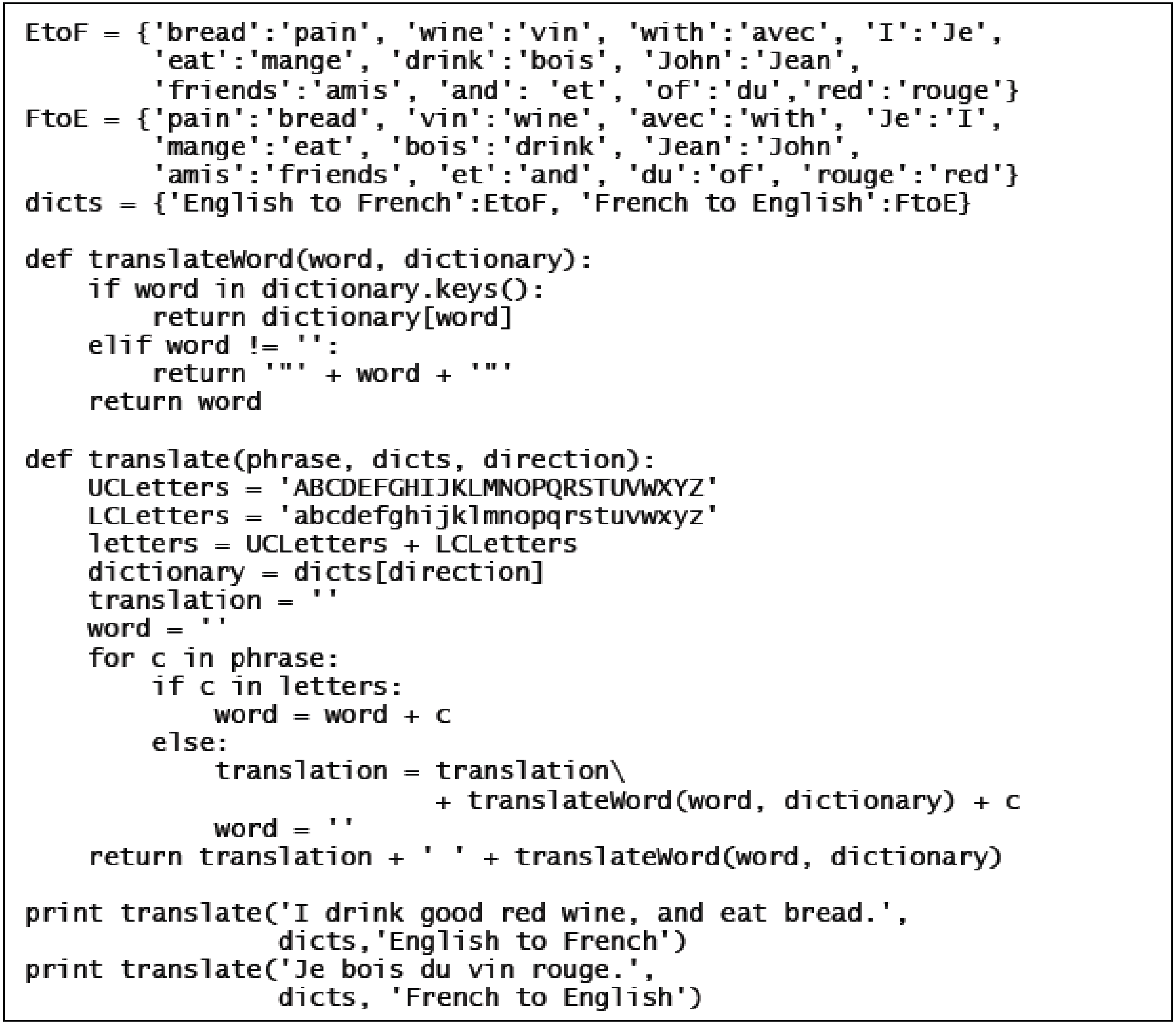
Dictionaries…

- Most programming languages do not contain a built-in type that provides a mapping from keys to values.
- Instead, programmers use other types to provide similar functionality.
- It is, for example, relatively easy to implement a dictionary using a list in which each element is a key/value pair.
- One can then write a simple function that does the associative retrieval,

Dictionaries…

```
def keySearch(L, k):
        for elem in L:
                if elem[0] == k:
                        return elem[1]
        return None
```

Dictionaries…



```
EtoF = {'bread':'pain', 'wine':'vin', 'with':'avec', 'I':'Je',
        'eat':'mange', 'drink':'bois', 'John':'Jean',
        'friends':'amis', 'and': 'et', 'of':'du','red':'rouge'}
FtoE = {'pain':'bread', 'vin':'wine', 'avec':'with', 'Je':'I',
        'mange':'eat', 'bois':'drink', 'Jean':'John',
        'amis':'friends', 'et':'and', 'du':'of', 'rouge':'red'}
dicts = {'English to French':EtoF, 'French to English':FtoE}

def translateWord(word, dictionary):
    if word in dictionary.keys():
        return dictionary[word]
    elif word != '':
        return '"' + word + '"'
    return word

def translate(phrase, dicts, direction):
    UCLetters = 'ABCDEFGHIJKLMNOPQRSTUVWXYZ'
    LCLetters = 'abcdefghijklmnopqrstuvwxyz'
    letters = UCLetters + LCLetters
    dictionary = dicts[direction]
    translation = ''
    word = ''
    for c in phrase:
        if c in letters:
            word = word + c
        else:
            translation = translation\
                          + translateWord(word, dictionary) + c
            word = ''
    return translation + ' ' + translateWord(word, dictionary)

print translate('I drink good red wine, and eat bread.',
                dicts,'English to French')
print translate('Je bois du vin rouge.',
                dicts, 'French to English')
```

Dictionaries…

```
monthNumbers = {'Jan':1, 'Feb':2, 'Mar':3, 'Apr':4, 'May':5, 1:'Jan', 2:'Feb', 3:'Mar', 4:'Apr', 5:'May'}
keys = [ ]
for e in monthNumbers:
        keys.append(str(e))
print(keys)
keys.sort()
print(keys)
```

- Might print
- ['Jan', 'Mar', ‘2’, ‘3’, ‘4’, ‘5’, ‘1’, 'Feb', 'May‘, ‘Apr’].
- [‘1’, ‘2’, ‘3’, ‘4’, ‘5’, 'Apr', 'Feb', 'Jan', 'Mar', 'May'].

Dictionaries…

```
birthStones = {‘Jan’:’Garnet’, ‘Feb’:’Amethyst’,
‘Mar’:’acquamarine’, ‘Apr’:’Diamond’, ‘May’:’Emerald’}
months = birthStones.keys()
print(months)
birthStones[‘June’]=‘Pearl’
print(months)
```

- Might Print

```
Dict_keys([‘Jan’, ‘Feb’, ‘May’, ‘Apr’, ‘Mar’])
Dict_keys([‘Jan’, ‘Mar’, ‘June’, ‘Feb’, ‘May’, ‘Apr’])
```

Dictionaries…

- **len(d)** returns the number of items in d.
- **d.keys()** returns a list containing the keys in d.
- **d.values()** returns a list containing the values in d.
- **k in d** returns True if key k is in d.
- **d[k]** returns the item in d with key k.
- **d.get(k, v)** returns d[k] if k is in d, and v otherwise.
- **d[k] = v** associates the value v with the key k in d. If there is already a value associated with k, that value is replaced.
- **del d[k]** removes the key k from d.
- **for k in d** iterates over the keys in d.

Thank you!!!