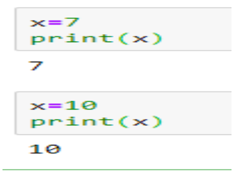
UNIT – III(a)

Iteration: Reassignment, Updating variables, The while statement, Break, Square roots, Algorithms.
Strings: A string is a sequence, len, Traversal with a for loop, String slices, Strings are immutable, Searching, Looping and Counting, String methods, The in operator, String comparison.

Iteration

Reassignment:

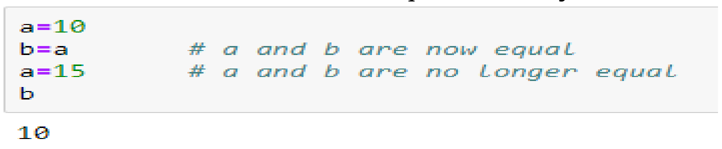
In Python it is possible to make more than one assignment to the same variable. A new assignment makes an existing variable refer to a new value (and stop referring to the old value).
Ex:



```
x=7
print(x)
7

x=10
print(x)
10
```

The first time we display x, its value is 7; the second time, its value is 10. Equality is a symmetric relationship and assignment is not. For example, in mathematics, if a = 7 then 7 = a. But in Python, the statement a = 7 is legal and 7 = a is not. Also, in mathematics, a proposition of equality is either true or false for all time. If a = b now, then a will always equal b. In Python, an assignment statement can make two variables equal, but they don't have to stay that way:



```
a=10
b=a          # a and b are now equal
a=15         # a and b are no longer equal
b

10
```

The third line changes the value of a but does not change the value of b, so they are no longer equal.

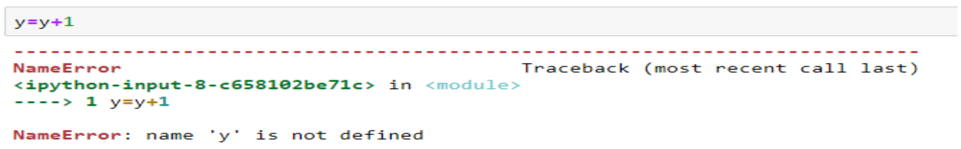
Updating Variables:

A common kind of reassignment is an update, where the new value of the variable depends on the old.

x = x + 1

This statement states that, "get the current value of x, add one, and then update x with the new value".

If you try to update a variable that doesn't exist, you get an error, because Python evaluates the right side before it assigns a value to x:



```
y=y+1

---------------------------------------------------------------------------
NameError                                 Traceback (most recent call last)
<ipython-input-8-c658102be71c> in <module>
----> 1 y=y+1

NameError: name 'y' is not defined
```

Before you can update a variable, you have to **initialize** it with a simple assignment:

```
y=0
y=y+1
print(y)
```

```
1
```

Note: Updating a variable by adding 1 is called an ***increment***; subtracting 1 is called a ***decrement***.

The *While* Statement:

In a computer program, repetition is also called iteration. Here repetition means repeating identical or similar tasks without making errors. Python provides while statement to make it easier.

Syntax:

```
while condition:
        statements
```

The below is the flow of execution for a while statement:

1. Determine whether the condition is true or false.
2. If false, exit the while statement and continue execution at the next statement.
3. If the condition is true, run the body and then go back to step 1.

This type of flow is called a loop because the third step loops back around to the top. The body of the loop should change the value of one or more variables so that the condition becomes false eventually and the loop terminates. Otherwise the loop will repeat forever, which is called an infinite loop.

#Python Script to find the sum of digits of a given number.

Source Code:

```
#Demonstration on usage of while loop - Sum of digits of a given number
def sumofDigits(n):
    sum=0
    while(n>0):
        rem=n%10
        sum+=rem
        n=int(n/10)
    return sum
num=int(input("Enter a number"))
print("Sum of digits of a given number is : ",sumofDigits(num))
```

Output:

```
Enter a number234
Sum of digits of a given number is :  9
```

break:

Sometimes you don't know it's time to end a loop until you get half way through the body. In that case you can use the break statement to jump out of the loop.

For example, suppose you want to take input from the user until they type "#". You could write:

```
while True:
    line = input(">")
    if line == "#":
        break
    print(line)
print("Task Completed!")
```

The loop condition is True, which is always true, so the loop runs until it hits the break statement. Each time through, it prompts the user with an angle bracket. If the user types #, the break statement exits the loop. Otherwise the program echoes whatever the user types and goes back to the top of the loop. See the below sample output:

```
>Hello II CSE Welcome to Python Programming Online Classes#
Hello II CSE Welcome to Python Programming Online Classes#
>#
Task Completed!
```

Square roots:

Loops are often used in programs that compute numerical results by starting with an approximate answer and iteratively improving it. For example, one way of computing square roots is Newton's method. Suppose that you want to know the square root of a. If you start with almost any estimate, x, you can compute a better estimate with the following formula:

$$y = \frac{x + a/x}{2}$$

For example, if a is 4 and x is 3:

```
#Square root of a number
a=4
x=3
y = (x + a/x) / 2
y
```

```
2.1666666666666665
```

The result is closer to the correct answer (i.e square root of 4 = 2). If we repeat the process with the new estimate, it gets even closer:

```
#Square root of a number(continued)
x = y
y = (x + a/x) / 2
y
```

```
2.0064102564102564
```

After a few more updates, the estimate is almost exact:

```
#Square root of a number(continued)
x = y
y = (x + a/x) / 2
y
```

```
2.0000102400262145
```

```
#Square root of a number(continued)
x = y
y = (x + a/x) / 2
y
```

```
2.0000000000262146
```

In general, we don't know how many steps it takes to get to the right answer, but we know when we get there because the estimate stops changing:

```
#Square root of a number(continued)
x = y
y = (x + a/x) / 2
y
```

```
2.0
```

```
#Square root of a number(continued)
x = y
y = (x + a/x) / 2
y
```

```
2.0
```

When y == x, we can stop. Here is a loop that starts with an initial estimate, x, and improves it until it stops changing:

```
a=4
x=3
while True:
     y = (x + a/x) / 2
     if y == x:
          break
     x = y
print(x)
```

```
2.0
```

Rather than checking whether x and y are exactly equal, it is safer to use the built-in function abs to compute the absolute value, or magnitude, of the difference between them:

```
if abs(y-x) < epsilon:
        break
```

Where epsilon has a value like 0.0000001 that determines how close is close

Algorithms:

Newton's method is an example of an algorithm: it is a mechanical process for solving a category of problems (Here, computing square roots).

To understand what an algorithm is, it might help to start with something that is not an algorithm. When you learned to multiply single-digit numbers, you probably memorized the multiplication table. In effect, you memorized 100 specific solutions. That kind of knowledge is not algorithmic.

But if you were "lazy", you might have learned a few tricks. For example, to find the product of n and 9, you can write n-1 as the first digit and 10-n as the second digit. This trick is a general solution for multiplying any single-digit number by 9. That's an algorithm!

Similarly, the techniques you learned for addition with carrying, subtraction with borrowing, and long division are all algorithms. One of the characteristics of algorithms is that they do not require any intelligence to carry out. They are mechanical processes where each step follows from the last according to a simple set of rules.

Executing algorithms is boring, but designing them is interesting, intellectually challenging, and a central part of computer science.

Some of the things that people do naturally, without difficulty or conscious thought, are the hardest to express algorithmically. Understanding natural language is a good example. We all do it, but so far no one has been able to explain how we do it, at least not in the form of an algorithm.

Strings

A String is sequence:

A string is a sequence, which means it is an ordered collection of other values. A string is a sequence of characters. You can access the characters one at a time with the bracket operator:

```
str="Mango"
print(str[0])
print(str[2])
```

```
M
n
```

The second statement selects character positioned at index ‘0’. The third statement selects character positioned at index ‘2’. The expression in brackets is called an index. The index indicates which character in the sequence you want. The index is an offset from the beginning of the string, and the offset of the first letter is zero.

As an index you can use an expression that contains variables and operators:

```
i=3
print(str[i+1])
```

```
o
```

Note: The value of the index has to be an integer. Otherwise you get an error.

```
str[1.4]
```

```
---------------------------------------------------------------------------
TypeError                                 Traceback (most recent call last)
<ipython-input-12-89878cffe6df> in <module>
----> 1 str[1.4]

TypeError: string indices must be integers
```

len:

len() is a built-in function that returns the number of characters in a string:

```
branch="Computer Science & Engineering"
len(branch)
```

```
30
```

Suppose if you want to get the last letter of a string, generally you might try something like this:

```
length=len(branch)
branch[length]
```

```
---------------------------------------------------------------------------
IndexError                                Traceback (most recent call last)
<ipython-input-21-8defb267513e> in <module>
      1 length=len(branch)
----> 2 branch[length]

IndexError: string index out of range
```

The reason for the IndexError is that there is no letter in 'Computer Science & Engineering' with the index 30. Since we started counting at zero, the six letters are numbered 0 to 29. To get the last character, you have to subtract 1 from length:

```
branch[length-1]
```

```
'g'
```

Or you can use negative indices, which count backward from the end of the string. The expression branch[-1] yields the last letter, branch[-2] yields the second to last, and so on

```
print(branch[-1])
print(branch[-2])
```

```
g
n
```

Traversal with a *for* loop:

A lot of computations involve processing a string one character at a time. Often they start at the beginning, select each character in turn, do something to it, and continue until the end. This pattern of processing is called a traversal. One way to write a traversal is with a while loop:

```
branch="AI&DS"
index = 0
while index < len(branch):
    letter = branch[index]
    print(letter)
    index = index + 1
```

```
A
I
&
D
S
```

This loop traverses the string and displays each letter on a line by itself. The loop condition is index < len(branch), so when index is equal to the length of the string, the condition is false, and the body of the loop doesn't run. The last character accessed is the one with the index len(branch)-1, which is the last character in the string.

Another way to write a traversal is with a for loop:

```
for letter in branch:
    print(letter)
```

```
A
I
&
D
S
```

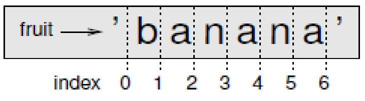


Figure : Slice indices.

Each time through the loop, the next character in the string is assigned to the variable letter. The loop continues until no characters are left.

String Slices:

A segment of a string is called a slice. Selecting a slice is similar to selecting a character:

```
branch="Computer Science"
print(branch[0:7])
print(branch[9:16])
```

```
Compute
Science
```

The operator [n:m] returns the part of the string from the "n-eth" character to the "m-eth" character, including the first but excluding the last.

If you omit the first index (before the colon), the slice starts at the beginning of the string. If you omit the second index, the slice goes to the end of the string:

```
branch[:4]
```

```
'Comp'
```

```
branch[4:]
```

```
'uter Science'
```

If the first index is greater than or equal to the second the result is an empty string, represented by two quotation marks:

```
fruit = 'Mango'
fruit[3:3]
```

```
''
```

```
fruit[:]
```

```
'Mango'
```

An empty string contains no characters and has length 0, but other than that, it is the same as any other string.

Strings are Immutable:

If we try to use the [] operator on the left side of an assignment, with the intention of changing a character in a string.

```
greet="Welcome Guys"
greet[2]='M'
greet
```

```
---------------------------------------------------------------------------
TypeError                                 Traceback (most recent call last)
<ipython-input-37-b1b9e0cae852> in <module>
      1 greet="Welcome Guys"
----> 2 greet[2]='M'
      3 greet

TypeError: 'str' object does not support item assignment
```

The “object” in this case is the string and the “item” is the character you tried to assign. The reason for the error is that strings are immutable, which means you can’t change an existing string. The best you can do is create a new string that is a variation on the original:

```
greet = 'Welcome Guys'
new_greet = greet[0:2]+'M'+greet[3:]
new_greet
```

```
'WeMcome Guys'
```

Searching:

Traversing a sequence and returning when we find what we are looking for, is called a **search**.

#Python Script to demonstrate searching

Source Code:

```
#Searching Demonsration

def find(word, letter):
    index = 0
    while index < len(word):
        if word[index] == letter:
            return index
        index = index + 1
    return -1
word=input("Enter a word\n")
letter=input("Enter a letter to search in the given word ")
res=find(word,letter)
if res==-1:
    print(letter," not found in ",word)
else:
    print(letter," found at position ",res,"in ",word)
```

Output:

```
Enter a word
Hellocse
Enter a letter to search in the given word p
p  not found in  Hellocse

  Enter a word
  Hellocse
  Enter a letter to search in the given word e
  e  found at position  1 in  Hellocse
```

find() is the inverse of the [] operator. Instead of taking an index and extracting the corresponding character, it takes a character and finds the index where that character appears. If the character is not found, the function returns -1. If word[index]== letter, the function breaks out of the loop and returns immediately. If the character doesn’t appear in the string, the program exits the loop normally and returns -1.

Looping and Counting:

The following program counts the number of times a specific letter appears in a given string:

#Python script to count no. of times a letter appears in a given word.

Source Code:

```python
def count(word, letter):
    index = 0
    count=0
    for let in word:
        if let == letter:
            count = count + 1
    return count
word=input("Enter a word\n")
letter=input("Enter a letter to search in the given word ")
res=count(word,letter)
if res==0:
    print(letter," not found in ",word)
else:
    print(letter," appeared ",res,"times in ",word)
```

```
Enter a word
welcome
Enter a letter to search in the given word H
H  not found in  welcome
```

```
Enter a word
Welcome
Enter a letter to search in the given word e
e  appeared  2 times in  Welcome
```

This program demonstrates another pattern of computation called a counter. The variable ***count*** is initialized to ***0*** and then incremented each time an **‘*a*’** is found. When the loop exits, count contains the result—the total number of a’s.

String methods:

Strings provide methods that perform a variety of useful operations. A method is similar to a function—it takes arguments and returns a value—but the syntax is different.

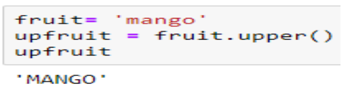
String Method	Purpose
upper()	Takes a string and returns a new string with all uppercase letters.
find()	find substrings, not just characters. By default, find starts at the beginning of the string, but it can take a second argument, the index where it should start

The method ***upper()*** takes a string and returns a new string with all uppercase letters. Instead of the function syntax upper(word), it uses the method syntax ***word.upper()***.

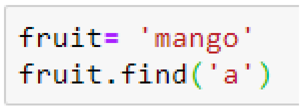
This form of dot notation specifies the name of the method, upper, and the name of the string to apply the method to, word. The empty parentheses indicate that this method takes no arguments.

A method call is called an invocation; in this case, we would say that we are invoking upper on word.

Ex :



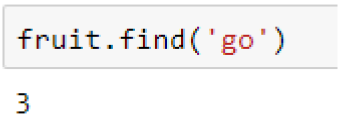
```
fruit= 'mango'
upfruit = fruit.upper()
upfruit
'MANGO'
```



```
fruit= 'mango'
fruit.find('a')
```

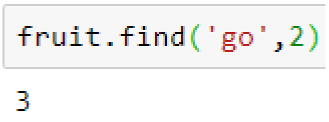
In this example, ***find()*** is invoked on **fruit** and the letter we are looking for is passed as a parameter.

It can also find substrings, not just characters:



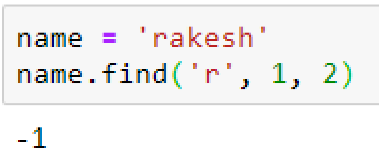
```
fruit.find('go')
3
```

By default, find starts at the beginning of the string, but it can take a second argument, the index where it should start:



```
fruit.find('go',2)
3
```

This is an example of an **optional argument**; find can also take a third argument, the index where it should stop:

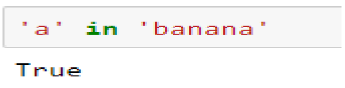


```
name = 'rakesh'
name.find('r', 1, 2)
-1
```

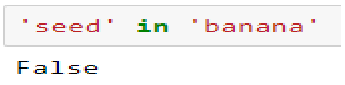
This search fails because ***r*** does not appear in the index range from 1 to 2, not including 2. Searching up to, but not including, the second index makes find consistent with the slice operator.

The *in* operator:

The word ***in*** is a boolean operator that takes two strings and returns **True** if the first appears as a substring in the second:



```
'a' in 'banana'
True
```



```
'seed' in 'banana'
False
```

The following function prints all the letters from word1 that also appear in word2:

```
def in_both(word1, word2):
    for letter in word1:
        if letter in word2:
            print(letter)
word1=input("Enter first word : ")
word2=input("Enter first word : ")
in_both(word1, word2)
```

Here’s what you get if you compare apples and oranges:

```
Enter first word : apples
Enter first word : oranges
a
e
s
```

String comparison:

The relational operators work on strings. To see if two strings are equal:

```
word=input("Enter a word : ")
if word == 'banana':
    print('All right, bananas.')
```

```
Enter a word : banana
All right, bananas.
```

```
Enter a word : Mango
```

Other relational operations are useful for putting words in alphabetical order:

```
word=input("Enter a word : ")
if word < 'banana':
    print('Your word, ' + word + ', comes before banana.')
elif word > 'banana':
    print('Your word, ' + word + ', comes after banana.')
else:
    print('All right, bananas.')
```

```
Enter a word : pineapple
Your word, pineapple, comes after banana.
```

```
Enter a word : banana
All right, bananas.
```

```
Enter a word : mango
Your word, mango, comes after banana.
```

Python does not handle uppercase and lowercase letters the same way people do. All the uppercase letters come before all the lowercase letters, so:

Your word, Pineapple, comes before banana.

A common way to address this problem is to convert strings to a standard format, such as all lowercase, before performing the comparison.

UNIT – III(b)

Case Study: Reading word lists, Search, Looping with indices.
Lists: List is a sequence, Lists are mutable, Traversing a list, List operations, List slices, List methods, Map filter and reduce, Deleting elements, Lists and Strings, Objects and values

Case Study

Reading word lists:

There are lots of word lists available on the Web, but the one most suitable for our purpose is one of the word lists collected and contributed to the public domain by Grady Ward. It is a list of 113,809 official crosswords; that is, words that are considered valid in crossword puzzles and other word games. In the Moby collection, the filename is 113809of.fic; you can download a copy, with the simpler name words.txt, from http://thinkpython2.com/code/words.txt.

This file is in plain text, so you can open it with a text editor, but you can also read it from Python. The built-in function open takes the name of the file as a parameter and returns a file object you can use to read the file.

fin = open('words.txt')

fin is a common name for a file object used for input. The file object provides several methods for reading, including ***readline***, which reads characters from the file until it gets to a newline and returns the result as a string:

fin.readline()
'aa\n'

The first word in this particular list is “aa”, which is a kind of lava. The sequence \n represents the newline character that separates this word from the next. The file object keeps track of where it is in the file, so if you call ***readline*** again, you get the next word:

fin.readline()
'aah\n'

The next word is “aah”, which is a perfectly legitimate word, so stop looking at me like that. Or, if it’s the newline character that’s bothering you, we can get rid of it with the string method strip:

```
line = fin.readline()
word = line.strip()
word
'aahed'
```

You can also use a file object as part of a for loop. This program reads words.txt and prints each word, one per line:

```
fin = open('words.txt')
```

for line in fin:
word = line.strip()
print(word)

Search:

The simplest search example is:

```
def has_no_e(word):
    for letter in word:
        if letter == 'e':
            return False
    return True
word=input("Enter a word")
has_no_e(word)
```

The for loop traverses the characters in word. If we find the letter “e”, we can immediately return False; otherwise we have to go to the next letter. If we exit the loop normally, that means we didn’t find an “e”, so we return True.

avoids is a more general version of has_no_e but it has the same structure:

```
def avoids(word, forbidden):
    for letter in word:
        if letter in forbidden:
            return False
    return True
```

We can return **False** as soon as we find a forbidden letter; if we get to the end of the loop, we return **True**.

uses_only is similar except that the sense of the condition is reversed:

```
def uses_only(word, available):
    for letter in word:
        if letter not in available:
            return False
    return True
```

Instead of a list of forbidden letters, we have a list of available letters. If we find a letter in word that is not in available, we can return **False**.

uses_all is similar except that we reverse the role of the word and the string of letters:

```
def uses_all(word, required):
    for letter in required:
        if letter not in word:
            return False
    return True
```

Instead of traversing the letters in word, the loop traverses the required letters. If any of the required letters do not appear in the word, we can return **False**.
One can observe that ***uses_all*** was an instance of a previously solved problem, and you would have written:
def uses_all(word, required):
return

Looping with indices:

For is_abecedarian we have to compare adjacent letters, which is a little tricky with a for loop:

```
def is_abecedarian(word):
    previous = word[0]
    for c in word:
        if c < previous:
            return False
        previous = c
    return True
```

An alternative is to use recursion:

```
def is_abecedarian(word):
    if len(word) <= 1:
        return True
    if word[0] > word[1]:
        return False
    return is_abecedarian(word[1:])
```

Another option is to use a while loop:

```
def is_abecedarian(word):
    i = 0
    while i < len(word)-1:
        if word[i+1] < word[i]:
            return False
        i = i+1
    return True
```

The loop starts at i=0 and ends when i=len(word)-1. Each time through the loop, it compares the ith character (which you can think of as the current character) to the i + 1th character (which you can think of as the next). If the next character is less than (alphabetically before) the current one, then we have discovered a break in the abecedarian trend, and we return False.

If we get to the end of the loop without finding a fault, then the word passes the test. To convince yourself that the loop ends correctly, consider an example like 'flossy'. The length of the word is 6, so the last time the loop runs is when i is 4, which is the index of the second-to-last character. On the last iteration, it compares the second-to-last character to the last, which is what we want.

Here is a version of is_palindrome that uses two indices; one starts at the beginning and goes up; the other starts at the end and goes down.

```
def is_palindrome(word):
    i = 0
    j = len(word)-1
    while i<j:
        if word[i] != word[j]:
            return False
        i = i+1
        j = j-1
    return True
word=input("Enter a word : ")
is_palindrome(word)
```

```
Enter a word : LIRIL

True
```

LISTS

List is a sequence:

Like a string, a list is a sequence of values. In a string, the values are characters; in a list, they can be any type. The values in a list are called elements or sometimes items. There are several ways to create a new list; the simplest is to enclose the elements in square brackets ([and]):

Ex:

[10, 20, 30, 40]

['cse', 'ece', 'ai&ds']

The first example is a list of four integers. The second is a list of three strings. The elements of a list don't have to be the same type. The following list contains a string, a float, an integer, and (lo!) another list:

['ece', 2.0, 5, [10, 20]]

A list within another list is nested. A list that contains no elements is called an empty list; you can create one with empty brackets, [].

You can also assign list values to variables:

```
#Lists Creation
branches = ['cse', 'ece', 'ai&ds']
numbers = [10,20,30,40]
emptylist = []
print(branches,numbers,emptylist)
```

```
['cse', 'ece', 'ai&ds'] [10, 20, 30, 40] []
```

Lists are mutable:

The syntax for accessing the elements of a list is the same as for accessing the characters of a string—the bracket operator. The expression inside the brackets specifies the index. Remember that the indices start at 0:

```
branches[0]
```

```
'cse'
```

Unlike strings, lists are mutable. When the bracket operator appears on the left side of an assignment, it identifies the element of the list that will be assigned.

```
numbers
```

```
[10, 20, 30, 40]
```

```
numbers[1]=45
print(numbers)
```

```
[10, 45, 30, 40]
```

The one-eth element of numbers, which used to be 20, is now 45. The below Figure shows the state diagram for branches, numbers and emptylist:

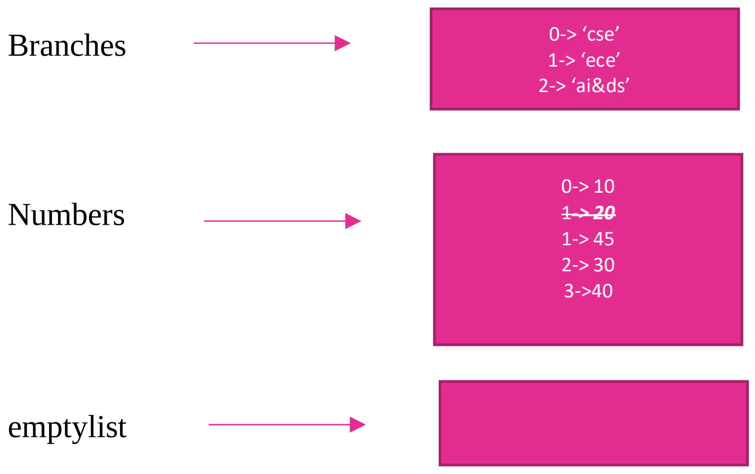


Fig: State Diagram

Lists are represented by boxes with the word "list" outside and the elements of the list inside. ***branches*** refers to a list with three elements indexed 0, 1 and 2. ***numbers*** contains 4 elements; the diagram shows that the value of the second element has been reassigned from 20 to 45. ***emptylist*** refers to a list with no elements.

List indices work the same way as string indices:

- Any integer expression can be used as an index.
- If you try to read or write an element that does not exist, you get an ***IndexError.***
- If an index has a negative value, it counts backward from the end of the list.

The in operator also works on lists.

```
branches = ['cse', 'ece', 'ai&ds']
'ece' in branches
```

```
True
```

```
'civil' in branches
```

```
False
```

Traversing a list:

The most common way to traverse the elements of a list is with a for loop. The syntax is the same as for strings:

```
for branch in branches:
    print(branch)
```

```
cse
ece
ai&ds
```

This works well if you only need to read the elements of the list. But if you want to write or update the elements, you need the indices. A common way to do that is to combine the built-in functions range and len:

```
print(numbers)
```

```
[10, 45, 30, 40]
```

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

```
[20, 90, 60, 80]
```

This loop traverses the list and updates each element. ***len()*** returns the 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 the next element. The assignment statement in the body uses i to read the old value of the element and to assign the new value. A for loop over an empty list never runs the body:

```
for x in []:
    print('This never happens.')
```

Although a list can contain another list, the nested list still counts as a single element. The length of this list is four:

```
len(['python', 10, ['cse', 'ece', 'ai&ds'], [10, 23, 35]])
```

```
4
```

List Operations:

The ***"+ operator"*** concatenates lists:

```
list1=[10,20,30,40]
list2=[5,15,25,35]
clist=list1 + list2
print(clist)
```

```
[10, 20, 30, 40, 5, 15, 25, 35]
```

The ***"* operator"*** repeats a list a given number of times.

```
[25]*4
```

```
[25, 25, 25, 25]
```

```
['cse',34]*2
```

```
['cse', 34, 'cse', 34]
```

The first example repeats [25] four times. The second example repeats the list ['cse',34] two times.

List Slices:

The slice operator also works on lists:

```
t = ['a', 'b', 'c', 'd', 'e', 'f']
print(t[1:3])
print(t[:4])
print(t[3:])
```

```
['b', 'c']
['a', 'b', 'c', 'd']
['d', 'e', 'f']
```

If you omit the first index, the slice starts at the beginning. If you omit the second, the slice goes to the end. So if you omit both, the slice is a copy of the whole list.

```
In [35]: t[:]

Out[35]: ['a', 'b', 'c', 'd', 'e', 'f']
```

Since lists are mutable, it is often useful to make a copy before performing operations that modify lists.

A slice operator on the left side of an assignment can update multiple elements:

```
t = ['a', 'b', 'c', 'd', 'e', 'f']
t[1:3] = ['x', 'y']
t
```

```
['a', 'x', 'y', 'd', 'e', 'f']
```

List Methods:

Python provides methods that operate on lists. For example, ***append()*** adds a new element to the end of a list:

```
t = ['a', 'b', 'c']
t.append('d')
t
```

```
['a', 'b', 'c', 'd']
```

extend() takes a list as an argument and appends all of the elements. This example leaves t2 unmodified.

```
l1 = ['a', 'b', 'c']
l2 = ['d', 'e']
l1.extend(l2)
print(l1)
print(l2)
```

```
['a', 'b', 'c', 'd', 'e']
['d', 'e']
```

sort() arranges the elements of the list from low to high:

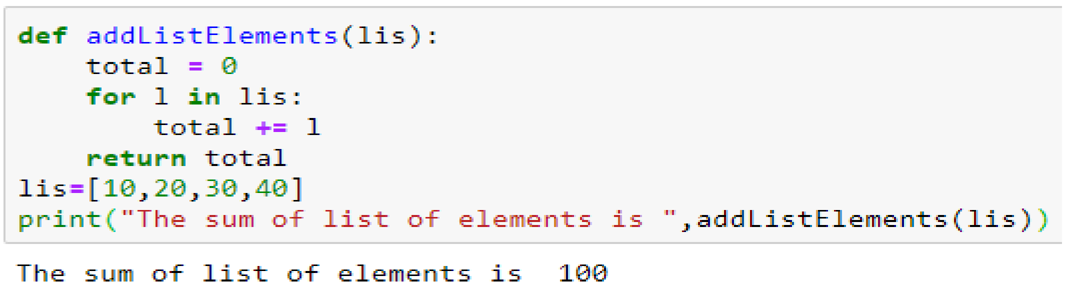
```
t = ['d', 'c', 'e', 'b', 'a']
t.sort()
t
```

```
['a', 'b', 'c', 'd', 'e']
```

Most list methods are void; they modify the list and return None.

Map, filter and reduce:

To add up all the numbers in a list, you can use a loop like this:



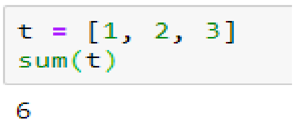
```
def addListElements(lis):
    total = 0
    for l in lis:
        total += l
    return total
lis=[10,20,30,40]
print("The sum of list of elements is ",addListElements(lis))
```

```
The sum of list of elements is  100
```

total is initialized to 0. Each time through the loop, ***l*** gets one element from the list. The += operator provides a short way to update a variable. This augmented assignment statement, ***total* += *l*** is equivalent to ***total* = *total* + *l***

As the loop runs, total accumulates the sum of the elements; a variable used this way is sometimes called an accumulator.

Adding up the elements of a list is such a common operation that Python provides it as a built-in function, sum:

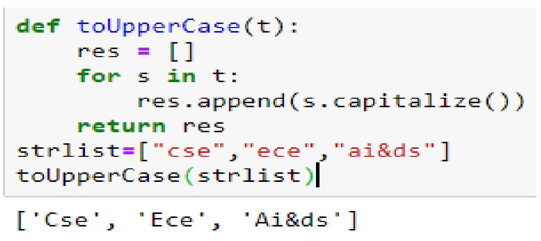


```
t = [1, 2, 3]
sum(t)
```

```
6
```

An operation like this that combines a sequence of elements into a single value is sometimes called ***reduce***.

Sometimes you want to traverse one list while building another. For example, the following function takes a list of strings and returns a new list that contains capitalized strings:



```
def toUpperCase(t):
    res = []
    for s in t:
        res.append(s.capitalize())
    return res
strlist=["cse","ece","ai&ds"]
toUpperCase(strlist)
```

```
['Cse', 'Ece', 'Ai&ds']
```

res is initialized with an empty list; each time through the loop, we append the next element. So ***res*** is another kind of accumulator.

An operation like ***toUpperCase()*** is sometimes called a ***map*** because it "maps" a function onto each of the elements in a sequence.

Another common operation is to select some of the elements from a list and return a sub-list.

For example, the following function takes a list of strings and returns a list that contains only the uppercase strings:

```
def only_upper(t):
    res = []
    for s in t:
        if s.isupper():
            res.append(s)
    return res
strlist=["cse","ECE","ai&ds"]
only_upper(strlist)
```

```
['ECE']
```

isupper is a string method that returns ***True*** if the string contains only upper case letters. An operation like ***only_upper()*** is called a filter because it selects some of the elements and filters out the others.

Note: Most common list operations can be expressed as a combination of map, filter and reduce.

Deleting Elements:

There are several ways to delete elements from a list. If you know the index of the element you want, you can use ***pop()***:

```
charlist = ['a', 'b', 'c']
x = charlist.pop(1)
print(charlist)
print(x)
```

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

pop() modifies the list and returns the element that was removed. If you don't provide an index, it deletes and returns the last element. If you don't need the removed value, you can use the ***del*** operator:

```
charlist = ['a', 'b', 'c']
del charlist[1]
charlist
```

```
['a', 'c']
```

If you know the element you want to remove (but not the index), you can use ***remove()***:

```
charlist = ['a', 'b', 'c']
charlist.remove('b')
charlist
```

```
['a', 'c']
```

The return value from remove is None. To remove more than one element, you can use ***del*** with a ***slice*** index:

```
charlist = ['a', 'b', 'c', 'd', 'e', 'f']
del charlist[1:5]
charlist
```

```
['a', 'f']
```

As usual, the slice selects all the elements up to but not including the second index.

Lists and Strings:

A string is a sequence of characters and a list is a sequence of values, but a list of characters is not the same as a string. To convert from a string to a list of characters, you can use list:

```
string = 'spam'
lis = list(string)
lis
```

```
['s', 'p', 'a', 'm']
```

The list function breaks a string into individual letters. If you want to break a string into words, you can use the ***split()*** method:

```
branch = 'computer science & engineering'
branchlis = branch.split()
branchlis
```

```
['computer', 'science', '&', 'engineering']
```

An optional argument called a delimiter specifies which characters to use as word boundaries. The following example uses a hyphen as a delimiter:

```
branch = 'computer,science &,engineering'
branchlis = branch.split(',')
branchlis
```

```
['computer', 'science &', 'engineering']
```

join() is the inverse of split. It takes a list of strings and concatenates the elements. ***join*** is a string method, so you have to invoke it on the delimiter and pass the list as a parameter:

```
t = ['computer','science','&','engineering']
delimiter = ' '
s = delimiter.join(t)
s
```

```
'computer science & engineering'
```

In this case the delimiter is a space character, so join puts a space between words. To concatenate strings without spaces, you can use the empty string, '', as a delimiter.

Objects and Values:

If we run these assignment statements:

```
a = 'banana'
b = 'banana'
```

We know that ***a*** and ***b*** both refer to a string, but we don't know whether they refer to the same string. There are two possible states, as shown in below figure.

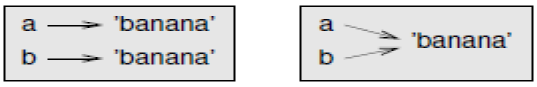


Figure : State diagram.

In one case, ***a*** and ***b*** refer to two different objects that have the same value. In the second case, they refer to the same object. To check whether two variables refer to the same object, you can use the is operator.

```
a = 'banana'
b = 'banana'
a is b
```

```
True
```

In this example, Python only created one string object, and both a and b refer to it. But when you create two lists, you get two objects:

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

```
False
```

So the state diagram looks like as shown in below figure.

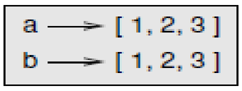


Figure : State diagram.

In this case we would say that the two lists are equivalent, because they have the same elements, but not identical, because they are not the same object. If two objects are identical, they are also equivalent, but if they are equivalent, they are not necessarily identical. Until now, we have been using "object" and "value" interchangeably, but it is more precise to say that an object has a value. If you evaluate [1, 2, 3], you get a list object whose value is a sequence of integers. If another list has the same elements, we say it has the same value, but it is not the same object.