

Fundamentals of Python

Part 2

(Reference: Fundamentals of Python, K.A Lambert and B.L Juneja)

The **list** And **dictionary**

- From the previous topics, we know that a **string** is a data structure that organizes text as a sequence of characters.
- In this section, we explore the use of two other common data structures:
 - The **list**, and
 - The **dictionary**
- A **list** allows the programmer to manipulate a sequence of data values of any types.
- A **dictionary** organizes data values by association with other data values rather than by sequential position.

The **list** Data Structure

- A **list** is a sequence of data values called **items** or **elements**.
- An **item** can be of any type:
 - A shopping list for the grocery store.
 - A to-do list
 - A guest list for a wedding
 - A recipe list for cooking
- The logical structure of a **list** is similar to the structure of a **string**. Each of the items in a **list** is ordered by position, like a character string.
- Each item in a **list** has a unique **index** that specifies its position.
- The **index** of the first item is 0 and the last item is **length of the list minus 1**

list Literals and Basic Operators

- In Python, a **list** is written as a sequence of data values separated by commas. The entire sequence is enclosed in square brackets ([and]).
- Here are the examples of such lists:
 - [1961, 1987, 1977, 1956, 1963] # a list of integers
 - ["Apple", "Orange", "Mango"] # a list of strings
 - [] # an empty list
- You can also creating a list of lists. Here is one example of such a list:
 - [[5, 9], [543, 341]] # list of lists
- It is interesting that when the Python interpreter evaluates a **list literal**, each of the elements is evaluated as well.

list Literals and Basic Operators

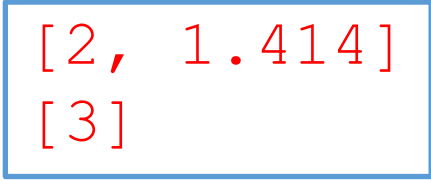
- When an element is a number or a string, that literal is included in the resulting list.
- When the element is variable or any other expression, its value is included in the list as shown below:

```
import math
```

```
x = 2
```

```
print([x, math.sqrt(x)])
```

```
print([x + 1])
```



```
[2, 1.414]  
[3]
```

list Literals and Basic Operators

- You can also build lists of integers using the **range** and **list** functions:

```
first = [1, 2, 3, 4, 5]
second = list(range(1, 6))
print(first)
print(second)
```

[1, 2, 3, 4, 5]
[1, 2, 3, 4, 5]

- The function **len** and the subscript operator **[]** work just as they do for strings:

```
first = [1, 2, 3, 4, 5]
print(len(first)) → 5
print(first[0]) → 1
print(first[2:4]) → [3, 4]
```

list Literals and Basic Operators

- **Concatenation (+)** and **equality (==)** also work as expected for **lists**:

```
first = [1, 2, 3, 4, 5]
```

```
second = first + [6, 7, 8]
```

```
print(second) → [1, 2, 3, 4, 5, 6, 7, 8]
```

```
print(first == second) → False
```

- The **print** function strips the quotation marks from a string, but it does not alter the list form:

```
print("12345") → 12345
```

```
print([1, 2, 3, 4, 5]) → [1, 2, 3, 4, 5]
```

list Literals and Basic Operators

- You can use the **in** operator to detect the presence or absence of a given element:

```
print(3 in [1, 2, 3, 4])    → True  
print(3 in [10, 22, 30])   → False
```

- Similarly, by using **in** operator with **index** method of the list object, we can extract the **index** of a list element:

```
myList = [10, 22, 34, 56, 81]  
if 22 in myList:  
    index = myList.index(22)  
    print(index)           → 1
```


Some Operators and functions were used with **list** (L refers to a **list**)

Operator Function	What it does
L[<an integer expression>]	Subscript used to access an element at the given index position.
L[<start>:<end>]	Slice for a sub list. Returns a new list.
L + L	List concatenation. Returns a new list.
print(L)	Prints the literal representation of the list.
len(L)	Returns the number of elements in a list
list (range(<upper>))	Returns a list containing the integers in the range 0 through upper-1
==, !=, <, >, >=, <=	Returns True if all the results are true, or False otherwise.
For<variable> in L:	Iterate through the list, binding variable to each element.
<any value> in L	Returns True if the value is in the list or False otherwise.

Replacing an Element in a list

- There is a huge difference between a string and a list:
 - A **string** is **immutable**, its structure and contents cannot be changed.
 - But a **list** is changeable, that is it is **mutable**; at any point in its life time, elements can be inserted, removed, or replaced.
 - The **list** itself maintains its identity, but its **state** – its length, and its contents can change.
- The **subscript** operator is used to replace an element at a given position, as shown below:

```
myList = [10, 22, 34, 56, 81]
myList[1] = 11
print(myList) → [10, 22, 34, 56, 81]
```

Replacing an Element in a list

- The following code shows how to replace each element of a list with its square:

```
myList = [2, 3, 4, 5]
index = 0
while index < len(myList):
    myList[index] = myList[index] ** 2
    index += 1
print(myList)  → [4, 9, 16, 25]
```

Replacing an Element in a list

- Using the string method **split** to extract a list of the words in a sentence. These words are then converted to uppercase letters within the list:

```
sentence = "This sentence has five words"
wordList = sentence.split()
index = 0
while index < len(wordList):
    wordList[index] = wordList[index].upper()
    index += 1
print(wordList)
```

```
['THIS', 'SENTENCE', 'HAS', 'FIVE', 'WORDS']
```

Replacing an Element in a **list**

- The next example replaces the first three elements of a **list** with new ones:

```
myNumber = list(range(6))      → [0, 1, 2, 3, 4, 5]  
myNumber[0:3] = [10, 11, 12]  
print(myNumber)               → [10, 11, 12, 3, 4, 5]
```

list Methods for Inserting and Removing Elements (L refers to a list)

List Method	What it does
L.append(element)	Adds element to the end of list L .
L.extend(aList)	Adds the elements of aList to the end of list L .
L.insert(index, element)	Inserts element at index if index is less than the length of L . Otherwise, insert element at the end of list L .
L.pop()	Removes and returns the element at the end of List L .
L.pop(index)	Removes and returns the element at index .

Examples of **list** Methods

- The method **append** expects just the new elements as an argument and adds the new element to the end of the **list**:

```
myList = [12, 34, 56, 89]
myList.append(97)
print(myList)           → [12, 34, 56, 89, 97]
```

- The method **insert** expects an integer **index** and the **new element** as arguments:

```
myList = [2, 3, 5, 6, 7]
myList.insert(3, 87)
print(myList)           → [2, 3, 5, 87, 6, 7]
```

- The **insert** method inserts an element in the given index by pushing the old element into the next index of the list.

Examples of list Methods

- The method **extend** performs a similar to that of **append** method, but it adds the **elements** of list argument **to the end of the list**:

```
myList = [2, 3, 5, 7]
myList.extend([9, 11, 13])
print(myList)           → [2, 3, 5, 7, 9, 11, 13]
```

- Concatenation of lists by using **extend** method:

```
myList1 = [2, 3, 5, 7]
myList2 = [9, 11, 13]
myList1.extend(myList2)
print(myList1)          → [2, 3, 5, 7, 9, 11, 13]
```


Examples of **list** Methods

- The method **pop** is used to remove an element at a given position. If the position is not specified, **pop** removes and **returns** the **last element** (not in a list form). If position is specified, **pop** removes the element at that position and **returns** it (not in a list form). In that case, the elements that followed the removed element are shifted one position to the left:

```
myList = [1, 2, 10, 11, 12, 13]
poppedList = myList.pop()
print(poppedList)                13
poppedList = myList.pop(2)
print(poppedList)                10
print(myList)                    [1, 2, 11, 12]
```

Searching a **list** Using **in** Operator

- After elements have been added to a list, a program can search for a given element in the list.
- The **in** operator determines an element's presence or absence with list **index** method:

```
myList = [12, 33, 45, 67, 78, 99]
key = 67
if key in myList:
    index = myList.index(key)
else:
    index = -1
if index >= 0:
    print("The key is in the index", index, "of the list.")
else:
    print("The key is not found!")
```

The key is in index 3 of the list.

Searching a **list** Using **sort** Operator

- A list method **sort** sorts a list by arranging its elements in ascending order.
- The **sort** method is a **mutable** object method, because such a method never returns a value to the caller (other methods such as **insert**, **append**, and **extend** are mutable list methods):

```
myList = [12, 9, 85, 17, 58, 29]
myList.sort()
print(myList)           [9, 12, 17, 29, 58, 85]
```

- Find smallest and largest numbers from a number list.
- Find the median of a given number list.

Mutator Methods and the Value **None**

- **Mutable** objects (such as **lists**) have some methods devoted entirely to modifying the internal state of the object.
- Such methods are called **mutators**.
 - Examples of **mutator** methods are: **list**, **insert**, **append**, **extend**, and **sort**.
- Because a change of state is all that is desired, a **mutator** method usually returns no value to the **caller**.
- Python automatically returns the special value **None** even when a method does not explicitly return a value.

Mutator Methods and the Value **None**

- Suppose you forget that **sort** never returns a value to the caller, nevertheless you think that it builds and returns a new **sorted list**, then see the return result of the following code:

```
aList = [8, 1, 3, 4, 0, 21, 13, 15]
aList = aList.sort()
print(aList)    None
```

- How this happens?

```
print(aList.sort()) None
print(aList)        [0, 1, 3, 4, 8, 13, 15, 21]
```

Aliasing and Side Effects

- We have learned that **numbers** and **strings** are **immutable**, that is we cannot change their internal structure.
- However, because **lists** are **mutable**, you can replace, insert, or remove elements from a list.
- The **mutable** property of lists leads to some interesting phenomena, as shown below:

```
first = [8, 1, 3, 4, 21, 13, 15]
second = first
print(first)           [8, 1, 3, 4, 21, 13, 15]
print(second)          [8, 1, 3, 4, 21, 13, 15]
first[1] = 99
print(first)           [8, 99, 3, 4, 21, 13, 15]
print(second)          [8, 99, 3, 4, 21, 13, 15]
```

Aliasing and Side Effects

- In the above example, a single list object is created and modified using the **subscript** operator.
- When the second element of list named **first** is replaced, the second element of the list named **second** is also replaced.
- This type of change is known as a **side effect**.
- This happens because after the assignment **second = first**, the variables **first** and **second** refer to the exact same **list object**.
- They are **aliases** for the same object. This phenomenon is known as **aliasing**.

Aliasing and Side Effects

- **Aliasing** is not always a good thing when side effects are possible. Assignment creates an **alias** to the same object rather than a reference to a copy of the object.
- To prevent **aliasing**, you can create a new object and copy the contents of the original as shown below:

```
first = [11, 12, 34, 56, 78]
second = []
for element in first:
    second.append(element)
print(first)           [11, 12, 34, 56, 78]
print(second)          [11, 12, 34, 56, 78]
first[1] = 23
print(first)           [11, 23, 34, 56, 78]
print(second)          [11, 12, 34, 56, 78]
```


A Simpler way to copy a **list** by Slice operation

- A simpler way to copy a list without **aliasing** is to use a **slice** over all of the positions, as follows:

```
first = [11, 12, 34, 56, 78]
second = []
second = first[:] #slicing of list over all positions
print(first)      [11, 12, 34, 56, 78]
print(second)     [11, 12, 34, 56, 78]
first[0] = 9
print(first)      [9, 12, 34, 56, 78]
print(second)     [11, 12, 34, 56, 78]
```

Class Exercises

- Write a Python program which stores n integer marks (out of 100) in a list which are by a user until the user just press Enter key.
- Get n integer elements from the user until press the Enter key and show the median of the list.
- Get n integer marks (out of 100) from a user (until just press the Enter key) and show its lowest, highest and average marks.
- From the above exercise, check whether how many students have passed the exam based on the class average (students with marks equal to or above the average marks will be passed the exam).

Equality: Object Identity and Structural Equivalence

- The `==` operator returns **True** if the variables are aliases for the same object.
- Unfortunately, the `==` operator also returns **True** if the contents of two different objects are the same.
- The first relation is called **object identity**, whereas the second relation is called **structural equivalence**.
- The `==` operator has no way of distinguishing between these two types of relations.

Object Identity by **is** operator

- Python's **is** operator can be used to test for object identity. It returns **True** if the two operands refer to the exact same object, and it returns **False** if the operands refer to distinct objects (even if they are structurally equivalent). The following code shows the difference between **==** and **is** operators:

```
first = [20, 30, 40]
second = first
third = [20, 30, 40]
print(first == second)      True
print(first == third)       True
print(first is second)      True
print(first is third)       False
```

Exercises

- **Question 1 [15 points]:** From a positive integer list (the elements of the list are collected from user until the Enter key is pressed), find the even elements (if any) and convert them into their binary values. Otherwise, display "There is no even element in the list".

Enter a positive integer or press Enter to stop: 3
Enter a positive integer or press Enter to stop: 8
Enter a positive integer or press Enter to stop: 24
Enter a positive integer or press Enter to stop: 7
Enter a positive integer or press Enter to stop:
The positive integer list is [3, 8, 24, 7]
The binary equivalent of 8 is 1000
The binary equivalent of 24 is 11000

Enter a positive integer or press Enter to stop: 1
Enter a positive integer or press Enter to stop: 3
Enter a positive integer or press Enter to stop: 5
Enter a positive integer or press Enter to stop:
The positive integer list is [1, 3, 5]
The list has no even integer element!

- **Question 2 [10 points]:** Get a string from user and if any of its substring start with 'f' or 'F', then replace that word from the string with "f-word" and display the resulted string. Otherwise, inform that the string has no "f-word".

Enter a string with words: Have a funny day, fine man!
Have a f-word day, f-word man!

Enter a string with words: Hello, see you later!
There is no f-word in the string!

Detecting Duplicate Elements from a list

- The following code shows how to detect duplicate elements from a list:

```
mylist = [10, 20, 10, 30, 40, 50]
val = False
for i in range(len(mylist)):
    for j in range(i + 1, len(mylist)):
        if mylist[i] == mylist[j]:
            val = True
print(val) True
```

Removing Duplicate Elements from a list

- Removing duplicate elements from a list by comparing the elements of the original list with another two lists as shown below:

```
myList = [1, 2, 1, 2, 6, 5, 5, 3, 3, 4, 4, 5, 6, 1]
output = []
seen = []
for x in myList:
    if x not in seen:
        output.append(x)
        seen.append(x)
print(output)          [1, 2, 6, 5, 3, 4]
print(seen)            [1, 2, 6, 5, 3, 4]
```

Set in Python

- A **set** is an unordered collection of items (elements). Every element in a set should be unique (has no duplicate elements) and must be immutable.
- However, the set itself is mutable (we can add or remove items from it).
- A **set** in Python is created by placing all the elements inside curly brace {}, separated by comma or by using the python function **set()**.
- A **set** can have any number of items and they may be of different types (such as integer, float, tuple, string, etc):

```
mySet = {1, 2, 3, 4}  
print(mySet)      {1, 2, 3, 4}
```


Using **set()** Function with **add** Method

- In Python we can generate a set of elements by using **set()** function and **add** method:

```
myset = set()
for x in range(8):
    myset.add(x)
print(myset) {0, 1, 2, 3, 4, 5, 6, 7}
```

- Next we can remove or eliminate duplicate elements from a **list** by using **set()** function and set method **add**:
 - That is, use a **for loop** and check each element of the **list** in a **set**. If the **set** has not that element, then append it to a **new list**.
 - The code is shown in the next slide.

Removing Duplicate Elements from a list

- The following code shows the removal of duplicates from a list:

```
myList = [1, 2, 1, 2, 6, 5, 5, 3, 3, 4, 4, 5, 6, 1]
output = []
seen = set()
for x in myList:
    if x not in seen:
        output.append(x)
        seen.add(x)
print(output)          [1, 2, 6, 5, 3, 4]
print(seen)            {1, 2, 3, 4, 5, 6}
```

Removing Duplicate Elements from a list

- Another easy way to eliminate duplicate elements from a list is just convert the list into a set:

```
mylist = [1, 1, 2, 3, 2, 3, 4, 5, 5]
mylist = list(set(mylist))
print(mylist)                [1, 2, 3, 4, 5]
```

Tuples

- A **tuple** is a type of sequence that resembles a **list**, except that, unlike a **list**, a **tuple** is **immutable**. The following code shows the usage of tuple in Python:

```
fruits = ("apple", "banana")
print(fruits)                ('apple', 'banana')
nonVeg = ("fish", "chicken")
print(nonVeg)                ('fish', 'chicken')
food = fruits + nonVeg
print(food)                  ('apple', 'banana', 'fish', 'chicken')
veg = ["beans", "celery"]
veg = tuple(veg)
print(veg)                   ('beans', 'celery')
```

Design Programs with Functions

- After completing this section, you will be able to:
 - Explain why functions are useful in structuring code in a program.
 - Employ top-down design to assign tasks to functions.
 - Define a recursive function.
 - Define a function with required and optimal parameters.
- Strictly speaking **functions** are not necessary; it is possible to construct any algorithm using only Python's built-in operators and control statements.
- However, in any significant program, the resulting code would be extremely complex, difficult to verify, and almost impossible to maintain.

Functions as Abstraction Mechanisms

- An **abstraction** is a mechanism that hides complex details of a program and thus allows a person to view many things as just one thing.
- Likewise, effective designers must invent useful abstractions to control code complexity.
- The first way that functions serve as abstraction mechanisms is by eliminating redundant, or repetitious code.
- Let's look at a function named **sum**, which returns the sum of two integer values to the caller section of the program.

A Simple Function that Returns the **sum**

- Function definition in Python has the following syntax:

```
def function_name(argument1, argument2,..., argumentn):  
    function description  
    return value
```

- Here is the definition of the function **sum**, that returns the sum of two integers:

```
def sum(a, b) :  
    return a + b  
val1 = int(input("Enter the first integer: "))  
val2 = int(input("Enter the second integer: "))  
print("The sum is", sum(val1, val2))
```

A Simple Function that Returns the **sum**

- A function named **sum**, which returns the sum of the numbers within a given range of numbers:

```
def sum(lower, upper):  
    sum = 0  
    for x in range(lower, upper + 1):  
        sum += x  
    return sum  
  
L = int(input("Enter the first integer: "))  
U = int(input("Enter the second integer: "))  
print("The sum from",val1, "to", val2, "is", sum(L,U))
```

```
Enter the first integer: 10  
Enter the second integer: 100  
The sum including 10 and 100 is 5005
```


Defining Simple Functions

- Most of the functions used thus far expect one or more arguments and return a value.
- Let's define a function that expects a number as an argument and returns the **square** of that number:

```
def square(x):  
    return x * x
```

```
value = int(input("Enter a positive integer: "))  
print(square(value))
```

Defining Simple Functions

- A function named **average** that returns the average value of an **integer list**:

```
def average(myList):  
    sum = 0  
    for x in myList:  
        sum += x  
    return sum/len(myList)
```

```
myList = []  
while True:  
    val = input("Enter list elements or just press Enter: ")  
    if (val == ""):  
        break  
    myList.append(int(val))  
avg = average(myList)  
print("The average of list is", avg)
```

```
Enter list elements or press Enter: 2  
Enter list elements or press Enter: 2  
Enter list elements or press Enter: 2  
Enter list elements or press Enter:  
The average of list is 2.0
```

Defining Simple Functions

- A function named **average** that returns the average value of a **float list**:

```
def average(myList):  
    sum = 0.0  
    for x in myList:  
        sum += x  
    return sum/len(myList)
```

```
myList = []  
while True:  
    val = input("Enter list elements or press Enter: ")  
    if (val == ""):  
        break  
    myList.append(float(val))  
avg = average(myList)  
print("The average of list is %0.2f"% avg)
```

```
Enter list elements or press Enter: 22.3  
Enter list elements or press Enter: 44.5  
Enter list elements or press Enter: 56.7  
Enter list elements or press Enter:  
The average of list is 41.17
```

A Function that not Returns its Value

- Here is the code sample of a Python function that not returns its value to the caller:

```
def average(val):  
    sum = 0  
    for x in val:  
        sum += x  
    print("The average of list elements is", sum/len(val))
```

```
myList = []  
while True:  
    data = input("Enter elements of a list or press just Enter: ")  
    if data == "":  
        break  
    myList.append(int(data))  
print("")  
average(myList)
```

```
Enter elements of a list or press just Enter: 10  
Enter elements of a list or press just Enter: 10  
Enter elements of a list or press just Enter: 10  
Enter elements of a list or press just Enter:  
The average of list elements is 10.0
```

Boolean Functions

- A **Boolean function** usually tests its argument for the presence or absence of some property. The function returns **True** if the property is present, or **False** otherwise. The following function shows the use of the Boolean function **odd**, which test an integer number to see whether it is odd:

```
def odd(number):  
    if number % 2 == 1:  
        return True  
    else:  
        return False  
  
number = int(input("Input a positive integer: "))  
val = odd(number)  
if val:  
    print(number, "is an odd number.")  
else:  
    print(number, " is not an odd number.")
```

```
Input a positive integer: 21  
21 is an odd number.  
Input a positive integer: 18  
18  is not an odd number.
```

Design with **Recursive** Functions

- A **recursive function** is a function that calls itself.
- To prevent a function from repeating itself indefinitely, it must contain at least one **selection statement**.
- This statement examines a condition called a **base case** (the *upper* value, see the below code) to determine whether to stop or to continue with another **recursive step**.
- Let us examine how to convert an iterative algorithm to a **recursive function**.
- Here is a definition of a function **displayRange** that prints the numbers from a lower bound to an upper bound:

```
def displayRange(lower, upper) :  
    while(lower <= upper) :  
        print(lower)  
        lower += 1
```

Design with Recursive Functions

```
def displayRange(lower, upper):  
    while(lower <= upper):  
        print(lower)  
        lower += 1
```

- How would we go about converting this function to a recursive one?
 - The equivalent recursive function replaces the **loop** with a **selection statement**, and the **assignment statement** is replaced with a **recursive call** of the function:

```
def displayRange(lower, upper):  
    if lower <= upper:  
        print(lower)  
        displayRange(lower + 1, upper)
```

Design with **Recursive** Functions

- Although the syntax and design of the two functions are different, the same algorithmic process is executed.
- Each call of the **recursive function** visits the next number in the sequence, just as the loop does in the iterative version of the function.
- Most recursive functions expect at least one assignment. This data value is used to test for the **base case** that ends the recursive process, and also is modified in some way before each recursive step.
- The modification of the data value should produce a new data value that allows the function to reach the base case eventually.

Design with **Recursive** Functions

- Here is a recursive function which prints repeated value of a string based on the given integer number:

```
def prinrMany(strVal, intNumber):  
    if intNumber > 0:  
        print(strVal)  
        prinrMany(strVal, intNumber - 1)  
myStr = input("Input a string: ")  
intNum = int(input("Enter the number: "))  
prinrMany(myStr, intNum)
```

```
Input a string: fine  
Enter the number: 3  
fine  
fine  
fine
```

Recursive Function Returns Value

- Recursive function that returns a **Value**. The following **sum** function computes and returns the sum of the numbers between the two values.
- In the recursive case, **sum** returns 0 if **lower** exceeds **upper** (the base case). Otherwise, the function adds **lower** to the **sum** of **lower + 1**, and **upper** and returns the result:

```
def sum(lower, upper):  
    if lower > upper:  
        return 0  
    else:  
        return lower + sum(lower +1, upper)
```

```
low = int(input("Enter the lower value: "))  
upp = int(input("Enter the upper value: "))  
sum = sum(low, upp)  
print("The sum is ", sum)
```

```
Enter the lower value: 1  
Enter the upper value: 5  
The sum is 15
```

Construct Recursive Function from Recursive Definition

- A **recursive function** consists of equations that state what a value is for one or more base cases, and one or more recursive cases.
- For example, the **Fibonacci** sequence is a series of values with a recursive definition.
- The first and second numbers in the **Fibonacci** sequence are 1.
- Thereafter, each number in the sequence is the sum of its two predecessors, as follows:

1 1 2 3 5 8 13 21.....

Construct Recursive Function from Recursive Definition

- A **recursive definition** of the n^{th} **Fibonacci** number is the following:

$\text{Fib}(n) = 1$, when $n = 1$, or $n = 2$

$\text{Fib}(n) = \text{Fib}(n - 1) + \text{Fib}(n - 2)$ for all $n > 2$

- Based on this we can construct a recursive function that computes and returns the n^{th} **Fibonacci** number:

```
def Fib(n):  
    if n < 3:  
        return 1  
    else:  
        return Fib(n - 1) + Fib(n - 2)
```

Construct Recursive Function from Recursive Definition

- The factorial of a positive integer **n**, **fact(n)**, is defined recursively as follows:

$\text{fact}(n) = 1$, when $n = 1$

$\text{fact}(n) = n * \text{fact}(n - 1)$, otherwise

Define a recursive function **fact** that returns the factorial of a given positive integer.

- Explain what happens when the following recursive function is called with the value 4 as an argument:

```
def example(n):  
    if n > 0:  
        print(n)  
        example(n - 1)
```

Construct Recursive Function from Recursive Definition

- Explain what happens when the following recursive function is called with the value 4 as an argument:

```
def example(n):  
    if n > 0:  
        print(n)  
        example(n)  
    else:  
        example(n - 1)
```

Higher-order Functions – the **map** function

- The **map** function:

- suppose we have a **list** named **words** that contains strings that represent integers. We want to replace each string with the corresponding integer value, the **map** function easily accomplishes this:

```
words = ["20", "99", "231"]  
words = list(map(int, words))  
print(words)      [20, 99, 231]
```

- The **map** function supports only a sequence data format such as a **list**, a **tuple**, or a **string**.

Higher-order Functions – the **filter** function

- **Filtering:**

- A second type of higher-order function is called a **filter** function. In this case, a **predicate** function is applied to each value in a **list**. If the predicate is **True**, the value passes the test, and is added to a **filter object** (similar to a **map** object, in the previous section). Otherwise, the value is dropped from consideration. Here is the Python's **filter** function that is used to produce a list of odd numbers in another list:

```
def odd(n) :  
    return n % 2 == 1  
  
oddList = list(filter(odd, range(10)))  
print(oddList)    [1, 3, 5, 7, 9]
```


Dictionaries

- We have seen that the **lists** organize their elements by **position**. This mode of organization is useful when you want to locate the first element, the last element, or visit each element in a sequence.
- However, in some situations, the position of a datum in a structure is irrelevant. For example, you might need to lookup Mr. John's phone number but don't care where that number is in the phonebook.
- A **dictionary** organizes information by **association**, not position. For example, when you use a dictionary to lookup the definition of "mammal", you don't start at page 1; instead, you directly focus the words beginning with 'M'.
- In Computer Science, data structures organized by association are called **tables** or **association lists**.

Exercise

- Using **map** and **filter** functions; get a set of list elements (string elements which are converted into integers by using the **map** function) and show a resulted list with only **odd integer** elements (by using the **filter** function).

Dictionary Literals

- In Python, a **dictionary** associates a set of **keys** with **data values**.
- A Python **dictionary** is written as a sequence of **key/value pair** separated by commas.
- These pairs are called **entries**. The entire sequence of **entries** is enclosed in curly braces({ and }).
- A colon (':') separates a **key** and its **value**. Some example dictionaries:
 - `phoneBook = {'John': '476-1234', 'Lily': '564-3216', 'Tom': '765-2398'}`
 - 'John', 'Lily', 'Tom' are **keys** and '476-1234', '564-3216', and '765-2398' are values.
 - `personalData = {'Name': 'Lily', 'Age': 30, 'Occupation': 'Teacher'}`
 - 'Name', 'Age', and 'Occupation' are **keys** and 'Lily', 30, and 'Teacher' are values.
 - **emptyDictionary = {}**

Adding Keys and Replacing values

- Add a new **key/value pair** to a dictionary by using the **subscript operator []**:
- Syntax: **<a dictionary>[<a key>] = <a value>**
- The following code segments creates an empty dictionary and adds two new entries:

```
personalInfo = {} ← empty dictionary
personalInfo["Name"] = "Sandy" ← value
personalInfo["Age"] = 30 ← key
personalInfo["Job"] = "Teacher"
print(personalInfo)
```

{ 'Name' : 'Sandy' , 'Age' : 30 , 'Job' : 'Teacher' }

Adding Keys and Replacing values

- The **subscript** is also used to replace a value at an existing key, as follows:

```
personalInfo = {}  
personalInfo["Name"] = "Sandy"  
personalInfo["Age"] = 30  
personalInfo["Job"] = "Teacher"  
print(personalInfo) { 'Name': 'Sandy', 'Age': 30, 'Job': 'Teacher' }  
personalInfo["Job"] = "Manager"  
print(personalInfo) { 'Name': 'Sandy', 'Age': 30, 'Job': 'Manager' }
```

Accessing Values from a dictionary- **get** method

- You can also use the **subscript** to obtain the value associated with a **key** in a dictionary. If the **key** is not present in the dictionary, Python raises an error:

```
personalInfo = {'Name': 'Sandy', 'Age': 30, 'Job': 'Teacher'}  
print(personalInfo["Job"])    Teacher  
print(personalInfo["Hobby"])  KeyError: 'Hobby'
```

- If the existence of a **key** is uncertain, the method **get** can be used to test availability of its value. The method **get** expects two arguments, a possible **key** and a default value.
- If the **key** is in the dictionary, the associated value is returned. If the key is absent, the default value **None** will be returned:

```
personalInfo = {'Name': 'Sandy', 'Age': 30, 'Job': 'Teacher'}  
print(personalInfo.get("Hobby"))  None  
print(personalInfo.get("Hobby", None))  None  
print(personalInfo.get("Hobby", "Hello"))  Hello
```

Removing Keys from a Dictionary

- To **delete** an entry from a dictionary, one can remove its **key** using the method **pop**.
- This method expects a key and an optional default value as arguments.
- If the key is in the dictionary, it is removed, and its associated value is returned.
- Otherwise, the default value is returned.
- If **pop** method is used with just one argument, and this key absent from the dictionary, Python raises an error.

Removing Keys from a Dictionary

- The following code sample attempts to remove two keys from a dictionary and prints the values returned:

```
personalInfo = {'Name': 'Sandy', 'Age': 30, 'Job': 'Teacher'}  
print(personalInfo.pop("Hobby", None)) None  
print(personalInfo.pop("Job")) Teacher  
print(personalInfo) { 'Name': 'Sandy', 'Age': 30 }
```


Traversing a Dictionary

- When a **for loop** is used with a dictionary, the loop's variable is bound to each key in an unspecified order:

```
info = {'Name': 'Sandy', 'Age': 30, 'Job': 'Clerk'}  
for key in info:  
    print(key, ":", info[key])
```

Name : Sandy
Age : 30
Job : Clerk

- Alternatively, we can use the dictionary method **items()** to access a **list** of the dictionary's entries:

```
info = {'Name': 'Lily', 'Age': 30, 'Job': 'Clerk'}  
print(info.items())           dict_items([('Name', 'Lily'), ('Age', 30), ('Job', 'Clerk')])  
print(list(info.items()))     [('Name', 'Sandy'), ('Age', 30), ('Job', 'Clerk')]  
print(set(info.items()))      {('Name', 'Sandy'), ('Age', 30), ('Job', 'Clerk')}
```

Traversing a Dictionary

- Note that the entries of the dictionary are represented as **tuples** within the list. A **tuple** of variables can then access the **key** and **value** of each entry in this **list** within a **for loop** (key in an unspecified order):

```
info = {'Name': 'Sandy', 'Age': 30, 'Job': 'Teacher'}  
for (key, value) in info.items():  
    print(key, ":", value)
```



Job : Teacher
Name : Sandy
Age : 30

- On each pass through the **for loop**, the variables **key** and **value** within the **tuple** are assigned the **key** and **value** of the current entry in the **list**.

Traversing a Dictionary

- If a special ordering of the **keys** is needed, then we can obtain a **list** of keys using the **keys method** and process this list to rearrange the **keys**:

```
info = {'Name': 'Sandy', 'Age': 30, 'Job': 'Teacher'}  
theKeys = list(info.keys())  
print(theKeys)  ['Job', 'Name', 'Age']  
theKeys.sort() ← Sorting the keys by sort() method  
print(theKeys)  ['Age', 'Job', 'Name']  
for key in theKeys:  
    print(key, ":", info[key])
```

Age: 30
Job: Teacher
Name: Sandy

DICTIONARY OPERATION	WHAT IT DOES (where 'd' refers to a Dictionary)
<code>len(d)</code>	Returns the number of entries in d
<code>aDict[key]</code>	Used for inserting a new key, replacing a value, or obtaining a value at an existing key.
<code>d.get(key [, default])</code>	Returns the value if the key exists or returns the default if the key does not exist. Raises an error if the default is omitted and the key does not exist.
<code>d.pop(key [, default])</code>	Removes the key and returns the value if the key exists or returns the default if the key does not exist. Raises an error if the default is omitted and the key does not exist.
<code>list(d.keys())</code>	Returns a list of the keys .
<code>list(d.values())</code>	Returns a list of the values .
<code>list(d.items())</code>	Returns a list of tuples containing the keys and values for each entry.
<code>d.has_key(key)</code>	Returns True if the key exists or False otherwise.
<code>d.clear()</code>	Remove all the keys.
For key in d:	key is bound to each key in d in an unspecified order.
<code>max(d.values())</code>	Returns the maximum of the values (in numbers).

Dictionary acts as a Lookup Table

- **Lookup table dictionary:** Here is the definition of the lookup table dictionary which is required for hexadecimal to binary conversion. The lookup table dictionary is given below:

```
hexToBinaryTable = { '0' : '0000', '1' : '0001', '2' : '0010',  
                    '3' : '0011', '4' : '0100', '5' : '0101',  
                    '6' : '0110', '7' : '0111', '8' : '1000',  
                    '9' : '1001', 'A' : '1010', 'B' : '1011',  
                    'C' : '1100', 'D' : '1101', 'E' : '1110',  
                    'F' : '1111' }
```

Dictionary acts as a Lookup Table

```
def hexToBinary(number, table):  
    binary = ""  
    for digits in number:  
        binary += table[digits]  
    return binary  
hexa = input("Enter an Hexadecimal number: ")  
hexa = hexa.upper()  
binaryVal = hexToBinary(hexa, hexToBinaryTable)  
print("The binary of 0x" + hexa + " is " + binaryVal)
```

Enter an Hexadecimal number: abf01

The binary of 0xABF01 is 10101011111100000001

- In the above code, show a “wrong input” display, if the input string from the user is wrong (means the input is not an exact hexadecimal value)

Dictionary Application: Finding **mode** Value

- The **mode** of a list of values is the value that occurs most frequently.
- The following script shows how to find frequency of elements in a string list using a dictionary:

```
myList = ["HELLO", "BYE", "BYE", "SEE", "YOU", "TAKE", "CARE", "BYE"]
```

```
myDictionary = {}
```

```
for words in myList:
    value = myDictionary.get(words, None)
    if value == None:
        myDictionary[words] = 1
    else:
        myDictionary[words] = value + 1
print(myDictionary)
```

```
{ 'BYE': 3, 'TAKE': 1, 'SEE': 1, 'HELLO': 1, 'YOU': 1, 'CARE': 1 }
```

Dictionary Application: Finding **mode** Value

- The **dictionary** associates each unique word with the number of occurrence (frequency) in the list ('myList'), and the script uses the function **max**, to return the maximum integer value contained in a dictionary. Here is the code for the script:

```
maxVal = max(myDictionary.values())
print(maxVal)
for key in myDictionary:
    if myDictionary[key] == maxVal:
        print("The mode is \"" + key + "\"")
```

The mode is "BYE"

Exercises

- Assume that the variable **data** refers to the dictionary {"b":20, "a":35}. Write the values of the following expressions:
 - **data["a"]**
 - **data.get("c", None)**
 - **len(data)**
 - **data.keys()**
 - **data.values()**
 - **data.pop("b")**
 - **data**
- Find the **mode** value from a given text file (.txt file).

Text Files

- A **text file** is a software object that stores data on a permanent medium such as a disk.
- Using text editor such as **Notepad**, you can create, view, and save data in a text file.
- The data in a text file can be viewed as characters, words, numbers, or lines of text, depending on the text file's format.
- In Python, all data output to or input from a text file must be **strings**.
 - Thus, numbers must be converted to strings before output, and these strings must be converted back to numbers after input.

Writing Text to a File

- Data can be **output** (write) to a text file using a **file** object.
- Python's **open** function, which expects a ***fileName***, and ***file-mode*** as argument.
- The *file-mode* is **r** for **input files** and **w** for **output files**.
- Thus the following code opens a **file object** on a file named **myfile.txt** for output:

```
f = open ("myfile.txt", 'w')
```

- where **f** is the file object variable.
- If the file **does not exist**, it is created with the given file name.
- If the file **already exists**, Python opens it. When data are written to the file, and the file is closed, any data previously existing in the file are erased.

Writing Text to a File

- String data are written (output) to a file using the method **write** with the file object:

f.write(string)

- If you want the output text to end with a **newline**, you must include the escape character '**\n**' in the string:

f.write("First line \n second line\n.....\n")

- When all of the outputs are finished, the file should be closed using the method **close**, as follows:

f.close()

Writing numbers to a File

- The file method **write** expects a **string** as an argument.
- Therefore, other types of data, such as integers or floating-point numbers, must first be converted to strings before being written to an **output** file.
- The resulting strings are then written to a file with a space or a newline as a separator character. The next code segment illustrates the output of integers to a text file:

```
import random
f = open("integers.txt", 'w')
for count in range(500):
    number = random.randint(1, 500)
    f.write(str(number) + "\n")
f.close()
```

Reading Text From a File

- Open a file for **input** in a manner similar to opening a file for **output**.
- The only thing that changes is the **mode** argument, which is **'r'**.
- Here is the code for opening **myfile.txt** for input:

```
f = open ("myfile.txt", 'r')
```

- The simplest way to use the file method **read** to input the entire contents of the file as a single **string**.
- If the file contains multiple line of text, the newline characters will be embedded in the string. The following code shows how to use the method **read**:

```
f = open("integers.txt", 'r')  
text = f.read()  
print(text)
```

Reading Text From a File

- After input is finished, another call to **read** would return an **empty string**, to indicate that the end of the file has been reached.
- To repeat an input, the file must be re-opened.
- It is not necessary to **close** the file after a read operation.
- Alternatively, an application can read and process the text from a file by using a **for** loop:

```
f = open("integers.txt", 'r')  
for line in f:  
    print(line)
```

- This will print each line of text with an extra **newline**.

Reading Text From a File

- The print the read line with extra newline (in the previous case) can be solved with **readline** method.
- The **readline** method consumes a line of input and returns a string, including a newline. If **readline** encounters the end-of-file(EOF), it returns the empty string. The following code shows the usage of **readline**:

```
f = open("integers.txt", 'r')
while True:
    line = f.readline()
    if line == "":
        break
print(line)
```


Reading Numbers from a File

- When reading data from a file, another important consideration is the format of the data item in the file.
- Here use the string method **strip** to remove the **newline**. The sample code with **strip** method is shown below:

```
f = open("integers.txt", 'r')
sum = 0
for line in f:
    line = line.strip() # removes extra new line
    number = int(line)
    sum += number
print("The sum is", sum)
```

Reading Numbers from a File

- Obtaining numbers from a text file in which they are **separated by spaces**, you can use the string method **split** to obtain a list of strings representing integers. The following code shows the usage of **split** method:

```
f = open("integers.txt", 'r')
sum = 0
for line in f:
    wordList = line.split() # removes extra space
    for word in wordList:
        number = int(word)
        sum += number
print("The sum is", sum)
```

Some File Operations

Method	What it does
open(fileName, mode)	Opens a file with the given name and returns a file object. The mode 'r', 'w', 'rw' or 'a'. Where 'rw' means read/write, and 'a' means append.
f.close()	Closes the output file. But not needed for input files .
f.write(string)	Outputs (writes) a string to file.
f.read()	Inputs(reads) the contents of a file and returns them as a single string.
f.Readline()	Inputs a line of text and returns it as a string, including the newline. Returns an empty string if the EOF is reached.