UCL

INST0004 Programming 2

Lecture 09: Sets, Sorting & Searching

Module Leader:
Dr Daniel Onah

2023-24


UCL

Copyright Note

Important information to adhere to

Summary of Previous Lecture

Recap of previous lecture

In the last lecture we looked at the following ...

- basic concepts of object-oriented programming
- understanding more about classes and objects creation
- we discussed different types of constructors
- we discussed the concepts of inheritance in object-oriented programming
- we discussed the concepts of polymorphism
- we looked at the various built-in class functions

Learning Outcomes

The learning outcomes for the lecture

The objective of this week's lecture is to introduce more of the concepts of classes and objects in Python programming. At the end of the lecture, you should be able to:

- understand **Sets & its constructs**
- understand the concepts **Mutable and Immutable objects**
- understand more on **subsets** of a Set
- understand how to **implement** Set in Python
- understand **Sorting & Searching** in Python

Introduction to Sets

The idea behind Sets in Python ...

Ideally, the easiest way to organise multiple values in Python programming is to use a ***container***. Let's look at a few examples of built-in Python containers. Some examples are: ***list***, ***dictionary*** and ***set***. We will look at how to combine containers to model complex programming structures. When you write a program that *manages a collection of unique items*, **sets** are *far more efficient than* **lists**.

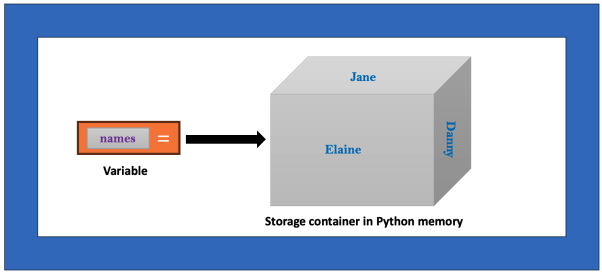
A **Set** is a container that stores a collection of unique values. Note that **set cannot contain duplicate elements**. Unlike a **list**, the elements or members of a set are *not stored in any particular order and cannot be accessed by the element position*. Set ***operations*** in Python programming is similar to the set theory and **operations** in Mathematics. Set operations are much more elegant and faster than the equivalent list operations.

Creating and Using Sets

Using Sets in Python ...

To create a set with initial elements, you can specify the elements **enclosed in braces**, similar to mathematics. For example, let's create a set of names.

```
names = {"Danny", "Elaine", "Jane"}
```



Remember: ***Set elements are not stored in any particular order.***

Creating and Using Sets

Using Sets in Python ...

Another way you could present the previous construct is to use a set function to convert the **sequence** or any list sequence into a set. Let's see how this is done.

```
names = ["Danny", "Elaine", "Jane"]
cast = set(names) # converting a list elements to set.
```

A set is created using a set literal or the set function.

A set is an unordered presentation of elements in no particular order.

Creating and Using Sets

Using Sets in Python ...

We **cannot use empty braces {}** to make an empty set in Python. Instead, we use the **set function with no arguments** to create an empty set:

```
cast = set()
```

Similar to any other Python containers used for collecting and storing elements, you can use the ***len()*** function to obtain the number of elements in a set:

```
numberOfCharacters = len(cast)
```

Using the “in” operator

Using “in” operator ...

The ***in*** operator is used to test whether an element is a member of a set. To determine whether an element is contained in the set, we use the *in* operator or its inverse, the ***not in*** operator:

```
#!/usr/bin/env python
"""
A program to demonstrate the ‘in’ operator
"""
names = ["Danny", "Elaine", "Jane"]
cast = set(names) # converting a list elements to set.
if "Jane" in cast:
    print("Jane is a member of the set.")
else:
    print("Jane is not a member of the set.")
```

Using the "For Loop" control structure

Using "for loop" with set ...

Because **sets** are *unordered*, you cannot access the elements of a set directly by using the position as you can with a ***list***. Instead, we use a ***for loop*** to iterate over the individual elements.

Note that the **order of the elements in the output** is *different from the order* in which the **set was originally created**.

```
1 #!/usr/bin/env python
2 """
3 A program to demonstrate the 'for loop' construct
4 """
5 names = ["Danny", "Elaine", "Jane"]
6 cast = set(names) # converting a list elements to set.
7 print("The names in the cast set includes:")
8 for name in cast:
9     print(name)
```

Using the "For Loop" control structure

Using "for loop" with set ...

As you can see in the output of the program, the **order** in which the elements of the set are **accessed** or **visited** depends on how they are originally stored internally. For example, the loop above displays the following output:

Output

```
The names in the cast set includes:
Elaine
Danny
Jane
```

Using the "For Loop" control structure

Using "for loop" with set ...

The fact that sets do not retain the initial ordering is not a problem when working with sets. *In fact, the lack of an ordering makes it possible to implement set operations very efficiently*. However, you would usually want to display the elements in **sorted order**. Use the ***sorted()*** function, this **returns a list** (not a set) of the elements in sorted order. The following loop prints the cast set in sorted alphabetical list order.

```
1 #!/usr/bin/env python
2 """
3 A program to demonstrate sorted set construct
4 """
5 names = ["Danny", "Elaine", "Jane"]
6 cast = set(names) # converting a list elements to set.
7 print("The names in the cast set includes:")
8 for name in sorted(cast): # sort the set in alphabetical order
9     print(name)
```

Using the "For Loop" control structure

Using "for loop" with set ...

You can notice that the elements of the set is then displayed in an **alphabetical order**. *Try to change the original entry of the names and see what happens*.

Output

```
The names in the cast set includes:
Danny
Elaine
Jane
```

Mutable & Immutable Objects

changeable and unchangeable ...

The value of some objects can change. Objects whose value can ***change*** *are said to be* ***mutable****; objects whose value is* ***unchangeable*** *once they are created are called* ***immutable****.* The value of an **immutable container** that contains a reference to a **mutable object** *can be changed if that mutable object is changed*. However, **the container is still considered immutable** because when we talk about the mutability of a container only the **identities of the contained objects are implied**.

Adding and Removing Elements from a Set

Using add and remove methods ...

Similar to **lists** and **dictionaries**, **sets** are **mutable collections**, *this means you can update the set by adding new elements and you can also remove elements from the set*. Mutable containers are containers for which the items stored in them can be updated. An immutable container are containers for which the elements stored in it **cannot be modified or changed**.

Adding Elements to a Set

Using add() method ...

Let's look at an example of how to add element to existing set using the ***add()*** method. We will also use the sorted() method to display the elements in the set in an alphabetical order.

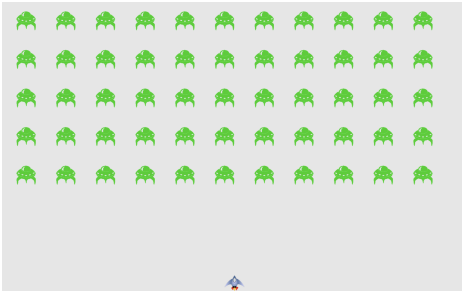
```
#!/usr/bin/env python
"""
A program ro demonstrate set add() method
"""
name = set(["Danny", "Elaine", "Jane"])
print(name)
name.add("Daniel")
print(name)
# use the sorted() method to display names alphabetically
for name in sorted(name):
    print(name)
```

Adding Elements to a Set

Using add() method ...

Remember, that a set cannot contain duplicate elements in its collection. *If the element being added is not already contained in the set*, ***this will be added to the set and the size of the set will increased by one***.

If you attempt to add an element that is already in the set, there would be no effect and the set will be unchanged.



Removing Elements from a Set using discard()

Using discard() method ...

There are two methods that can be used to **remove individual elements** from a set. The *discard()* method removes an element if the element exists in the set. However, this method **would have no effect if the given element** *is not a member of the set or does not exists in the set*.

```
#!/usr/bin/env python
"""
A program to demonstrate the discard() method
"""
name = {"Danny", "Elaine", "Jane", "Dan"}
print(name)
name.discard("Dan") # here the name 'Dan' is discarded
print(name)
# here let's try to discard a name no longer in the list
name.discard("Dan")
print(name)
```

Removing Elements from a Set using remove()

Using remove() method ...

The ***remove()***method, on the other hand, removes an element if it exists, but **raises an exception if the given element is not a member of the set**.

```
#!/usr/bin/env python
"""
A program to demonstrate the remove() method
"""
name = {"Danny", "Elaine", "Jane", "Dan"}
print(name)
name.remove("Dan") # here the name 'Dan' is removed
print(name)
# here trying to remove a name no longer in the list using the remove()
    method
name.remove("Dan") # this would lead to KeyError: 'Dan'
print(name)
```

Clear Elements of a Set using clear()

Using clear() method ...

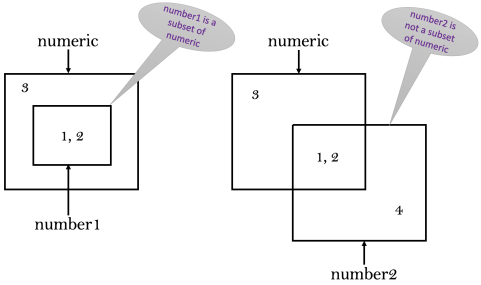
Finally, the ***clear()*** method removes all elements of a set, leaving it with an **empty set**.

```
1 #!/usr/bin/env python
2 """
3 A program to demonstrate the clear() method
4 """
5 name = {"Danny", "Elaine", "Jane", "Dan"}
6 print(name)
7 name.clear() # produces an empty set
8 print(name)
```

Subsets of a Set

What is a subset ...

A **set** is a *subset* of another **set** ***if and only if every element of the first set is also an element of the second set***. Where there is an intersection between the elements of two or more sets and they have a few common items or values, in this case we **do not have a subset**. *A subset in our case would be when the set is contained entirely within another set*.



A Set is a Subset if it is contained entirely within another Set.

issubset() Method

How do we use issubset() ...

The ***issubset()*** method tests whether one set is a subset of another set. This method returns **True** or **False** to ***ascertain whether one set is a subset of another***. Let's look at an example to help us understand how to construct this properly.

```
1 #!/usr/bin/env python
2 """
3 This program is to demonstrate issubset() methods
4 """
5 # decalre the sets
6 numeric = {1, 2, 3}
7 number1 = {1, 2}
8 number2 = {1, 2, 4}
```

issubset() Method

How do we use issubset() ...

```
# using conditional statement
if number1.issubset(numeric):
    print("************************* SUBSET *************************
    ")
    print("All number1 elements are in the numeric set elements!
    \ntherefore number1 is a subset of numeric.")
if not number2.issubset(numeric):
    print("************************* NOT SUBSET
    ************************")
    print("Not all the elements in number2 set belongs to numeric set!
    \ntherefore number2 is not a subset of numeric.")
```

issubset() Method

Output ...

```
************************ SUBSET ************************
All number1 elements are in the numeric set elements!
therefore number1 is a subset of numeric.
************************ NOT SUBSET ************************
Not all the elements in number2 set belongs to numeric set!
therefore number2 is not a subset of numeric.
```

Set Equality Operators (==, !=)

Equality operators ...

We can also use the equality operators == and != to test whether two sets are equal or not. Two sets are equal ***if and only if*** *they have exactly the same elements*.

Set Equality Operators (==, !=)

Equality operators ...

```
#!/usr/bin/env python
"""
A program to demonstrate set eqality operators
"""
numeric = {1, 2, 3}
number3 = {1, 2, 3}
number4 = {1, 2, 3, 4}
# conditional statement
if (numeric == number3):
    print("The numeric set and number3 set has the same digits")
if(numeric != number4):
    print("The numeric set and number4 set does NOT have the same digits
     in common")
```

Set *union()* Method

Set *union()* in Python ...

The **set union** is a *union of two sets containing all the elements from both sets, without any duplicates*. In Python, we use the set ***union()*** method to create the **union between two sets**.

```
#!/usr/bin/env python
"""
A program to demonstrate set union() method
"""
numeric = {3}
number = {1, 2, 4}
allElements = numeric.union(number) # this would display elements from
    both sets
print(allElements) # displays {1, 2, 3, 4}
```

Remember that the ***union()*** method returns a **new set**. *It does not modify either of the sets in the call*.

Set *intersection()* Method

Set *intersection()* in Python ...

The ***intersection of two sets contains all of the elements that are in both sets***. This intersection() method is used to display the elements that **both sets have or share in common**. We use the ***intersection()*** method to create the intersection of two Python sets.

```
#!/usr/bin/env python
"""
A program to demonstrate set intersection() method
"""
numeric = {1, 2, 3, 4}
number = {1, 2, 4}
inBothSets = numeric.intersection(number) # this would display elements
    that both set have in common
print(inBothSets) # displays {1, 2, 4}
```

Set *difference()* Method

Set *difference()* in Python ...

The difference of two sets results in a **new set** that *contains all the elements in the first set that are not in the second set*. The ***difference()*** method produces a new set with **the elements that belong to the first set but not the second** set.

```
#!/usr/bin/env python
"""
A program to demonstrate set difference() method
"""
numeric = {1, 2, 3, 4, 5}
number = {1, 2, 4}
# this would display elements that both set does not have in common ...
    but concentrating on numeric
diffSets = numeric.difference(number)
print(diffSets) # displays {3, 5}
```

Set *Union(), intersection() and difference()* Methods

Set *Python union(), intersection and difference() ...*

Note that when performing the set union or intersection of two sets, the **order does not matter**. For example, set **union()** can be performed in these ways.

```
numeric.union(number)
```

is the same set as:

```
number.union(numeric)
```

This would result to the same outcome.

Set *Union(), intersection() and difference()* Methods

Set *Python union(), intersection and difference() ...*

Note that when performing the set union or intersection of two sets, the **order does not matter**. For example, set **intersection()** can be performed in these ways.

```
numeric.intersection(number)
```

is the same set as:

```
number.intersection(numeric)
```

This would result to the same outcome.

Set *Union(), intersection() and difference()* Methods

Set *Python union(), intersection and difference() ...*

In the case of set **difference()** method, the **order of expression certainly matters**. For example the **outcome of the two expression below would set return different results**. Remember the **difference()** method ***concentrate mostly on the first set and considers the element in the first set first before comparing with the second set***.

```
1  #!/usr/bin/env python
2  """
3  A program to demonstrate set difference() method
4  """
5  numeric = {1, 2, 3, 4, 5}
6  number = {1, 2, 4}
7  diffSets1 = numeric.difference(number) # this would display elements
       that both set does not have in common ... but concentrating on
       numeric
8  print(diffSets1) # displays {3, 5}
9  diffSets2 = number.difference(numeric)
10 print(diffSets2) # displays an empty set - set{}
```

Set Example Problem

Set problem task ...

Let's consider a simple **set** task for our practical application example.

Program description:

The program should be able to read a file that contains correctly spelled words and places the words in a set. It then reads all words from a wrongly spelled text file into a second set. Finally, the program should display all the words from the wrongly spelled text file. This should display all words from the wrongly spelled words that are not in the set of the correctly spelled words. These are the potential misspellings. Let's look at how to write this program.

Firstly, ***consider and think of which set method would be appropriate for this task***.

Set Example Problem

Set problem task ...

Now create a text file to store all the correctly spelled word. Save this file as **correct-words.txt**. The words should be entered into the text file using delimiter (*i.e. each word should be separated with a comma*) and can be in any order or cases as below:

```
Hello, Mango, Peter, Program, WELCOME, HOUSE, biscuit
```

Now create a text file to store all the wrongly spelled word. Save this file as **wrong-words.txt**. Similar to the previous entry, the words should be entered into the text file using delimiter (*i.e. each word should be separated with a comma*) and can be in any order or cases as below:

```
lleoh, maOng, suoHe, EMLCOWE, BaCSUIST, teper, pgromar
```

Set Example Problem *Solution*

Set problem task solution ...

```
#!/usr/bin/env python
"""
This program checks which words in a wrongly spelled list that are not
    present in a list of correctly spelled words.
"""
# import split function from regular expression module
from re import split
def main():
    # read the word list and the document with wrongly spelled words
    correctSpelling = readWords("correctwords.txt")
    wrongSpelling = readWords("wrongwords.txt")
    # print all words in the wrongwords list that are not in the
    correctwords list
    misspellings = wrongSpelling.difference(correctSpelling)
    for word in sorted(misspellings):
        print(word)
```

Set Example Problem *Solution*

Set problem task solution ...

```
15 def readWords(filename):
16     setWord = set() # create an empty set to store the words
17     dataFile = open(filename, "r") # open files in read mode
18 
19     # iterate over the content of the files
20     for line in dataFile:
21         line = line.strip()
22 
23         # now use any character other than a-z or A-Z as word delimiters
       (to separate words) in the files.
24         subParts = split("[^a-zA-Z]+", line)
25 
26         for word in subParts:
27             if len(word) > 0:
28                 # add the word and make it lower case
29                 setWord.add(word.lower())
```

Set Example Problem *Solution*

Set problem task solution ...

```
30          # now close the file using the close() method
31          dataFile.close()
32
33          # return the words stored in the empty set created
34          return setWord
35
36  # start the program
37  main()
```

Set Example Problem *Counting Unique Words*

Set problem task unique word count...

Problem Statement A program to determine the number of unique words contained in a text document. *Our task is to write a program that reads the entire text document and determines the number of unique words in it or that are contained in the document without word duplication*.

Set Example Problem *Solution*

Set problem task solution ...

```
#!/usr/bin/env python
"""
A program to demonstrate counting the number of unique words in a file
"""
def main() :
    uniqueWords = set()
    filename = input("Enter filename (enter default as: article.txt): ")
    if len(filename) == 0 :
        filename = "article.txt"
    inputFile = open(filename, "r")
    for line in inputFile :
        theWords = line.split()
        for word in theWords :
            cleaned = clean(word)
            if cleaned != "" :
                uniqueWords.add(cleaned)
    print("The document contains", len(uniqueWords), "unique words.")
```

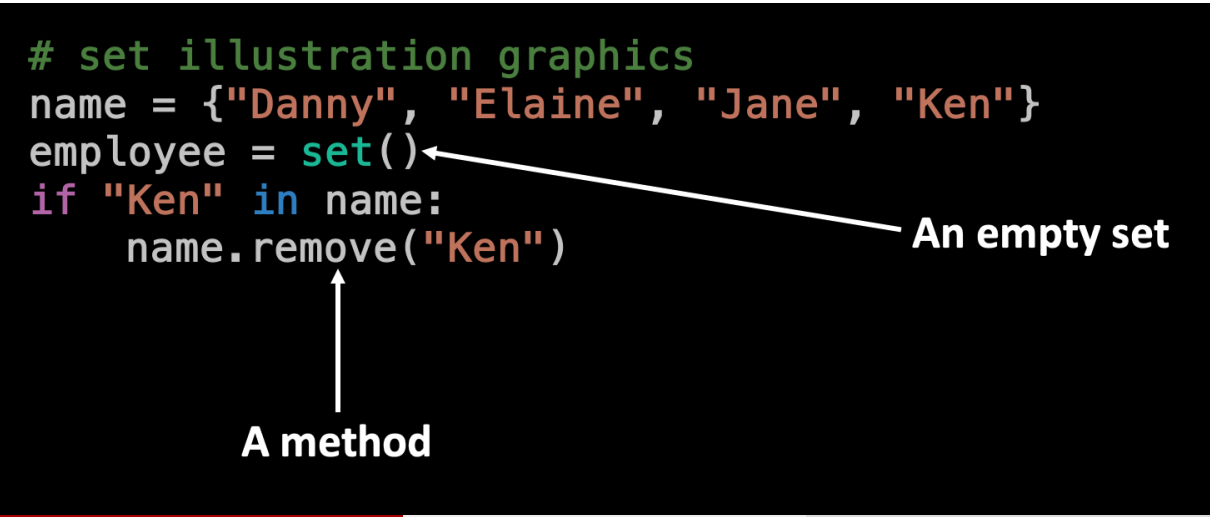
Set Example Problem *Solution*

Set problem task solution ...

```
18 # Cleans a string by making letters lowercase and removing characters
       that are not letters.
19 # @param string the string to be cleaned
20 # @return the cleaned string
21
22 def clean(string) :
23     result = ""
24     for char in string :
25         if char.isalpha() :
26             result = result + char.lower()
27             return result
28
29 # Start the program.
30 main()
```

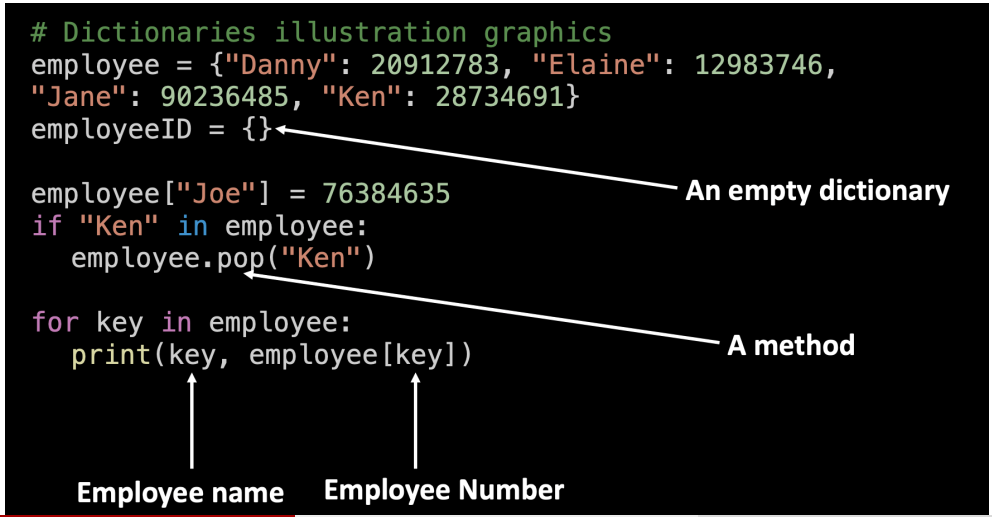
Key Points

Key points to remember about set...



Key Points

Key points to remember about dictionaries...




```
# Dictionaries illustration graphics
employee = {"Danny": 20912783, "Elaine": 12983746,
"Jane": 90236485, "Ken": 28734691}
employeeID = {}

employee["Joe"] = 76384635
if "Ken" in employee:
   employee.pop("Ken")

for key in employee:
   print(key, employee[key])
```

Sorting and Searching

What is sorting? ...

One of the most common tasks in data processing is **sorting** and **searching**. A **sorting algorithm** rearranges the elements of a collection so that they are stored in sorted order. For example, **we might be interested in displaying a list of employee in alphabetical order**. ***Once a list of elements is sorted, we can easily locate individual elements within the list***. In this section, we will be looking closely at several sorting methods to compare their performance. These *techniques would not just be useful for sorting algorithms, but also for analysing other algorithms*.

Sorting and Searching

What is sorting? ...

We will be looking at the following sorting and searching concepts:

- several sorting and searching algorithms
- show that algorithms for the same given tasks can differ widely in performance
- understand some important notations estimate and compare the performance of algorithms
- write code to measure the running time of a program

Selection Sort

What is selection sorting? ...

The selection sort algorithm **sorts a list** by ***repeatedly finding the smallest element of the unsorted tail region and moving it to the front***. We would firstly explore how to sort lists of integers, strings and more complex data.
A simple mechanism for selection sort. **In selection sort:**

- *pick the smallest element and swap it with the first one.*
- *Pick the smallest element of the remaining ones and swap it with the next one, and so on.*

Selection Sort Example

Selection sort example ...

```
#!/usr/bin/env python
"""
A program to demonstrate selection sort function algorithm
"""
# Sorting list using selection sort

def selectionSort(inputs):
    for n in range(len(inputs)):
        minPos = minimumPosition(inputs, n)
        temp = inputs[minPos] # swap the two elements
        inputs[minPos] = inputs[n]
        inputs[n] = temp
```

Selection Sort Example

Selection sort example ...

```
13 # find the smallest element in a tail range of the list
14 def minimumPosition(inputs, start):
15     minPos = start
16     for n in range(start + 1, len(inputs)):
17         if(inputs[n] < inputs[minPos]):
18             minPos = n
19     # return the position of the smallest element in the range
20     # values[start] . . . values[len(values) - 1]
21     return minPos
```

Selection Sort Example

Selection sort example ...

```
#!/usr/bin/env python
"""
A program to test the selection sort algorithm
"""
# This program demonstrates the selection sort algorithm by sorting a
# list that is filled with random numbers.
from random import randint
from selection_sort import selectionSort
# enter number to generate the selection sort
number = int(input("Enter an integer number:"))
inputs = []
```

Selection Sort Example

Selection sort example ...

```
12 for i in range(number):
13     inputs.append(randint(1, 100))
14 print("The random numbers generated")
15 print(inputs) # print randomly generated numbers up to 100
16 selectionSort(inputs) # sort the random numbers in ascending order
17 print("The randomly generated numbers sorted in ascending order")
18 print(inputs) # display the sorted random numbers
```

Selection Sort: Processing Time

Processing time of selection sort algorithm ...

Sometimes we will want to measure the duration or time taken for our program to finish the process of execution. In order to measure the performance of our program, ***we could traditionally or easily use a stop-watch device to measure how long it takes for the program to finish the execution***. ***However, most of programs that we write run very quickly, and it is not easy for us to time them accurately in this way using a stop-watch***. Furthermore, normally, **when a program takes a noticeable time to run, a certain amount of that time may simply be used for loading the program from disk into memory and displaying the result**. In order to measure the running time of an algorithm more accurately, **we will use the** *time()* library function from the **time module**. The *time module returns the seconds as a floating point value*. Let's look at an example to help us understand how to measure the sorting algorithm's performance.

Selection Sort: Processing Time

Processing time of selection sort algorithm example ...

```
#!/usr/bin/env python
"""
This program measures how long it takes to sort a list of a
user-specified size with the selection sort algorithm.
"""
from random import randint
from selection_sort import selectionSort
from time import time
# Prompt the user for the list size.
number = int(input("Enter list size: "))
```

Selection Sort: Processing Time

Processing time of selection sort algorithm example ...

```
# Construct random list.
inputValues = []
for i in range(number) :
    inputValues.append(randint(1, 100))
# create the start time
startTime = time()
selectionSort(inputValues)# selection sort execution process
# create the end time
endTime = time()
print("Elapsed time for the execution of the program is: %.3f seconds" %
     (endTime - startTime))
```

Note how the measurement for the start time and end time was placed within the program code construct. ***By starting to measure the time just before sorting, and stopping the timer just after, you get the time required for the sorting process, without counting the time for input and the output.***

Insertion Sort

What is insertion sort ...

Insertion sort is another sorting algorithm that places an **unsorted** element at its **suitable place in each iteration**. Insertion sort is another simple sorting algorithm. In this algorithm, we assume that the initial sequence of a list is already sorted. ***Insertion sort works in a similar way as we sort cards in our hand during a game of cards***. The big question here is ***How efficient is this algorithm?***.

Insertion sort has a desirable property:

- Its performance is O(n) if the list is already sorted.

This is a useful property in practical applications, ***in which data sets are often partially sorted***. To indicate that the number of visits is of order n^2, **computer scientists** often use **big-Oh notation**: Which indicate that the number of visits is $\mathbf{O}(n^2)$.

Insertion Sort

What is insertion sort ...

When the insertion sort algorithm starts, we initially set the value of **k to 0**. We then *enlarge the initial sequence by inserting the next list element*, **values[k + 1]**, at the proper location. When we reach the end of the list, ***the sorting process is complete***.

```
values[0]  values[1] ... values[k]
```

Insertion Sort

What is insertion sort ...

A SCENARIO:

Let's look at a typical scenario of an insertion sort. Let say $\boldsymbol{n}$ denote the size of the list. We carry out $\mathbf{n-1}$ iterations. In the k^{th} iteration, we have a sequence of **k elements** that is already sorted, and we need to insert a new element into the sequence. *For each insertion, we need to visit the elements of the initial sequence until we have found the location in which the new element can be inserted.* Then we need to move up the remaining elements of the sequence. Thus, the number of elements on the list visited would be $\mathbf{k+1}$ *(list elements are visited)*.

Insertion sort is the method that many people use to sort playing cards. Pick up one card at a time and insert it so that the cards stay sorted.

Insertion Sort Example

Example of insertion sort ...

```
#!/usr/bin/env python
"""
A program to demonstrate insertion sort
"""
## Sorts a list, using insertion sort.
# @param values the list to sort
def insertionSort(values) :
    for i in range(1, len(values)) :
        next = values[i]
    # Move all larger elements up.
        j= i
        while j > 0 and values[j - 1] > next :
            values[j] = values[j - 1]
            j=j- 1
    # Insert the element.
        values[j] = next
```

Insertion Sort Example

Example of insertion sort ...

```
#!/usr/bin/env python
"""
A program to test the insertion sort algorithm
"""
# This program demonstrates the selection sort algorithm by sorting a
# list that is filled with random numbers.
from random import randint
from insertionsort import insertionSort
# enter number to generate the selection sort
number = int(input("Enter an integer number:"))
inputs = []
```

Insertion Sort Example

Example of insertion sort ...

```
12 for i in range(number):
13     inputs.append(randint(1, 100))
14 print("The random numbers generated")
15 print(inputs) # print randomly generated numbers up to 100
16 insertionSort(inputs) # sort the random numbers in ascending order
17 print("The randomly generated numbers sorted in ascending order ")
18 print(inputs) # display the sorted random numbers
```

Merge Sort

What is merge sort ...

Merge sort algorithm, a **much more efficient algorithm than selection sort**. The basic idea behind merge sort is very simple. *Suppose we have a list of 10 integers. Let us engage in a bit of wishful thinking and hope that the first half of the list is already perfectly sorted, and the second half is too, like this*:

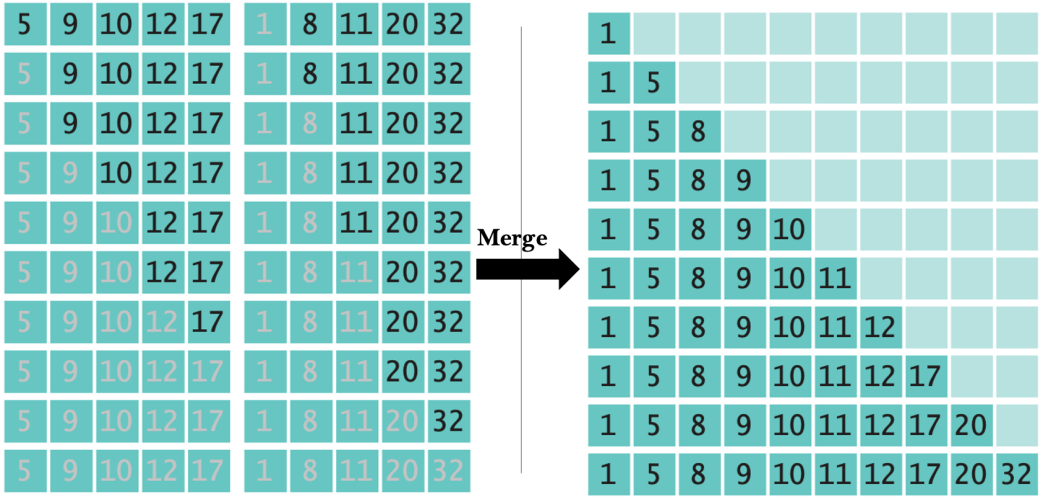
```
5 9 10 12 17 | 1 8 11 20 32
```

Now *it is simple to merge the two sorted lists into one sorted list*, by taking a new element from either the ***first*** or the ***second sublist***, and choosing the smaller of the elements each time.

The merge sort algorithm sorts a list by cutting the list in half, recursively sorting each half, and then merging the sorted halves.

Merge Sort

What is merge sort ...



Merge Sort

What is merge sort ...

Let's write a **mergesort.py** module that implements this idea. When the **mergeSort** function sorts a list, **it makes two lists**, *each half the size of the original*, and **sorts them recursively**. Then it **merges the two sorted lists together**.

Merge Sort Example

Example of merge sort ...

```
#!/usr/bin/env python
"""
A program to demonstrate merge sort procedure
"""
# The mergeSort function sorts a list, using the merge sort algorithm.
def mergeSort(inputValues):
    if len(inputValues) <= 1: return # take care not to give space between < and =
    middle = len(inputValues)// 2 # floor division (produce integer value with no remainder)
    first = inputValues[: middle] # start from begining to mid point
    second = inputValues[middle :] # start from mid point to end
    mergeSort(first)
    mergeSort(second)
    mergeListItems(first, second, inputValues)
```

Merge Sort Example

Example of merge sort ...

```
14 # Now let's merge two sorted lists into a third
15 def mergeListItems(first, second, inputValues):
16     nFirst = 0  # next element to consider in the first list
17     nSecond = 0 # next element to consider in the second list
18     k = 0       # next open position in the inputValues
19
20     # now as long as neither nFirst nor nSecond is past the end,
21     # move the smallest element into the inputValues
22     while nFirst < len(first) and nSecond < len(second):
23         if first[nFirst] < second[nSecond]:
24             inputValues[k] = first[nFirst]
25             nFirst = nFirst + 1
26         else:
27             inputValues[k] = second[nSecond]
28             nSecond = nSecond + 1
29         k = k + 1
```

Merge Sort Example

Example of merge sort ...

```
30 # Note here that only one of the two loops below copies entries
31
32     # copy any remaining entries of the first list.
33     while nFirst < len(first):
34         inputValues[k] = first[nFirst]
35         nFirst = nFirst + 1
36         k = k + 1
37
38     # copy any remaining entries of the second list.
39     while nSecond < len(second):
40         inputValues[k] = second[nSecond]
41         nSecond = nSecond + 1
42         k = k + 1
```

Merge Sort Example

Example of merge sort ...

```
#!/usr/bin/env python
"""
A program to test the merge sort algorithm by
sorting a list that contains random numbers.
"""
from random import randint
from mergesort import mergeSort
number = int(input("Enter an integer value: "))
inputValues = []
for i in range(number):
    inputValues.append(randint(1,100))
print("The random numbers generated")
print(inputValues)
mergeSort(inputValues)
print("The randomly generated numbers merged-sorted in ascending order")
print(inputValues)
```

QuickSort Algorithm

What is a quicksort ...

Quicksort is a commonly used algorithm that has the advantage over **merge sort** that no temporary lists are required to sort and merge the partial results. The **quicksort algorithm**, just as **merge sort**, is based on the strategy of divide and conquer. To sort a range **values[start] . . . values[to]** of the list values, ***first rearrange the elements in the range so that no element in the range values[start] . . . values[p]*** is larger than any element in the range **values[p + 1] . . . values[to]**. This step is called ***partitioning the range***.

QuickSort Example

Example of quicksort ...

```
#!/usr/bin/env python
"""
A program to demonstrate quick sort
"""
from partition import partition
def quickSort(values, start, end):
    if start >= end: return
    partit = partition(values, start, end)
    quickSort(values, start , partit)
    quickSort(values, partit + 1, end)
```

QuickSort Example

Example of quicksort ...

```
#!/usr/bin/env python
"""
A program to demonstrate partitioning of a quicksort algorithm
"""
def partition(values, start, end):
    pivot = values[start]
    i = start - 1
    j = end + 1
    while i < j:
        i = i + 1
        while values[i] < pivot:
            i = i + 1
        j = j - 1
```

QuickSort Example

Example of quicksort ...

```
14          while values[j] > pivot:
15              j = j - 1
16              if i < j:
17                  temp = values[i] # swap the two elements
18                  values[i] = values[j]
19                  values[j] = temp
20          return j
```

Searching

Searching in Python ...

Searching for an element in a list is an extremely common task. ***As with sorting, the right choice of algorithms can make a big difference in the searching task.***

Linear Search *"Sequential search"*

Linear search in Python ...

A linear search examines all values in a list until it finds a match or reaches the end of the list. **A linear search locates a value in a list in O(n) steps**. *Suppose, you want to find a number in a sequence of values in arbitrary order, and there is nothing you can do to speed up the search. One possible solution is to simply look through all elements until you found a match or until you reach the end*. This process is called a **linear** or **sequential search**. This is an algorithm used by Python's with operator when ***determining whether a given element is contained in a list***.

Linear Search

Linear search in Python ...

The big question is that *How long does a linear search take?* If we assume that the target **element is present in the list values**, then the average search visits ***n/2*** elements, where ***n*** *is the length of the list.* If it is not present, then all ***n*** elements must be inspected to verify the absence. Either way, a linear search is an **O(n)** algorithm. We will look at a function that performs linear searches through values or a list of integers. When searching for a target, the search function returns the first index of the match, or **-1** *if the target does not occur in values*.

Linear Search Example

Example of linear search ...

```
#!/usr/bin/env python
"""
A program to demonstrate linear search
"""
# This module implements a function for executing linear search in a list
# Finds a value in a list, using the linear search algorithms
def linearSearch(inputValues, target):
    for num in range(len(inputValues)):
        if inputValues[num] == target:
            return num
    return - 1
```

Linear Search Example

Example of linear search ...

```
#!/usr/bin/env python
"""
A program to test linear search algorithm
"""
from random import randint
from linearsearch import linearSearch
# construct random list
number = int(input("Enter an integer: "))
inputValues = []

# generate the random number based on the number of integer input
for i in range(number):
    inputValues.append(randint(1,100))

# display the random numbers to the number input
print(inputValues)
```

Linear Search Example

Example of linear search ...

```
13 finish = False
14 while not finish:
15     # enter an integer to search for the position
16     target = int(input("Enter number to search for (enter -1 to quit): "
       ))
17     if target == - 1:
18         finish = True
19     else:
20         position = linearSearch(inputValues, target)
21         if position == -1:
22             print("Number not found!")
23         else:
24             print("Number found in position: ", position)
```

Binary Search

What is a binary search ...

A binary search locates a value in a sorted list by determining whether the value occurs in the first or second half, then repeating the search in one of the halves. This search process is called a **binary search**, *because we cut the size of the search in half in each step*. That cutting in half works only because we know that the sequence of input values is sorted. We will look at a function that implements binary searches in a sorted list of integers. The **binarySearch** function returns the position of the match if the search succeeds, or **–1** ***if the target is not found in values***. Here, we will show a ***recursive version of the binary search algorithm***.

Binary Search

What is a binary search ...

Now let us search for a target in a data sequence that has been previously sorted. Of course, we could still do a linear search, but it turns out we can do much better than that.

```
4 7 9 2 5 8 9 6 10 26 39 76
```

*We would like to see whether the target **19** is in the data set.* **Let's narrow our search by finding whether the target is in the first or second half of the list**. The last value in the first half of the data set, *values[5]*, is **8**, which is smaller than the target. *Hence, we should look in the second half of the list for a match, that is, in the sequence.*

*Now let's determine the number of visits to list elements required to carry out a binary search. We can use the same technique as in the analysis of merge sort. Because we look at the middle element, which counts as one visit, and then search either the **left** or the **right sublist** for the target value.*

Binary Search

What is a binary search ...

```
#!/usr/bin/env python
"""
A program to demonstrate binary search algorithm
"""
# This module implements a function for executing binary searches in a list
# Finding a value in a range of sorted list, using the binary search algorithms
# return the index at which the target occurs, or -1 if it does not occur in the list

def binarySearch(inputValues, low, high, target):
    if low <= high:
        middle = (low + high)// 2
```

Binary Search

What is a binary search ...

```
12          if inputValues[middle] == target:
13              return middle
14          elif inputValues[middle] < target:
15              return binarySearch(inputValues, middle + 1, high, target)
16          else:
17              return binarySearch(inputValues, low, middle - 1, target)
18      else:
19          return -1
```

Note: *Because a binary search is so much faster than a linear search, is it worthwhile to sort a list first and then use a binary search? It depends right. If you search the list only once, then it is more efficient to pay for an O(n) linear search than for an O(n log(n)) sort and an O(log(n)) binary search. But if you will be making many searches in the same list, then sorting it is definitely worthwhile.*


Summary

Summary

Let's revise the concepts of today's lecture

In this lecture we discuss the following: Sets, Sorting & Searching algorithms ...

- Sets in Python and the various methods of creating set programs
- We discussed about Mutable and Immutable objects
- We looked at the various subset of Set with examples
- We discussed about **sorting** algorithms in Python
- We discussed about **searching** algorithms in Python

Summary

Let's revise the concepts of today's lecture

- Now you have understand how to implement **Sets** in Python programming. It has the same logic as **Dictionaries**, **Lists** and so on.
- We can use Sets built-in functions in a similar was as the other storage containers in Python programming.
- We want our programs to observe object-oriented programming paradigm by using the various constructs in a manner appropriate for the program we are developing.
- We can also apply this in enforcing information hiding and only allow access via **setter** and **getter** methods as the only means of modifying and retrieving information from the attributes.

Further Reading

chapters to find further reading on the concepts

You can read further this week's lecture from the following textbook chapters:

- Python for Everyone (3/e) : **By Cay Horstmann & Rance Necaise** - *Chapter 8 Sets & Dictionaries*, *Chapter 12 Sorting & Searching*
- Learning Python (5^{th} Edition): **By Mark Lutz** - *Chapter 4 Introducing Python Object Types* , *Chapter 5 Numeric Types*
- Python Programming for absolute beginner (3rd Edition): **By Michael Dawson** - *Chapter 5 Lists & Dictionaries: The Hangman Game*
- Python Crash Course - A hands-on, project-based introduction to programming (2nd Edition): **By Eric Matthes** - *Chapter 6 Dictionaries*

UCL

Next Lecture 10

In week 10

Lecture 10: Files & Exceptions Handling