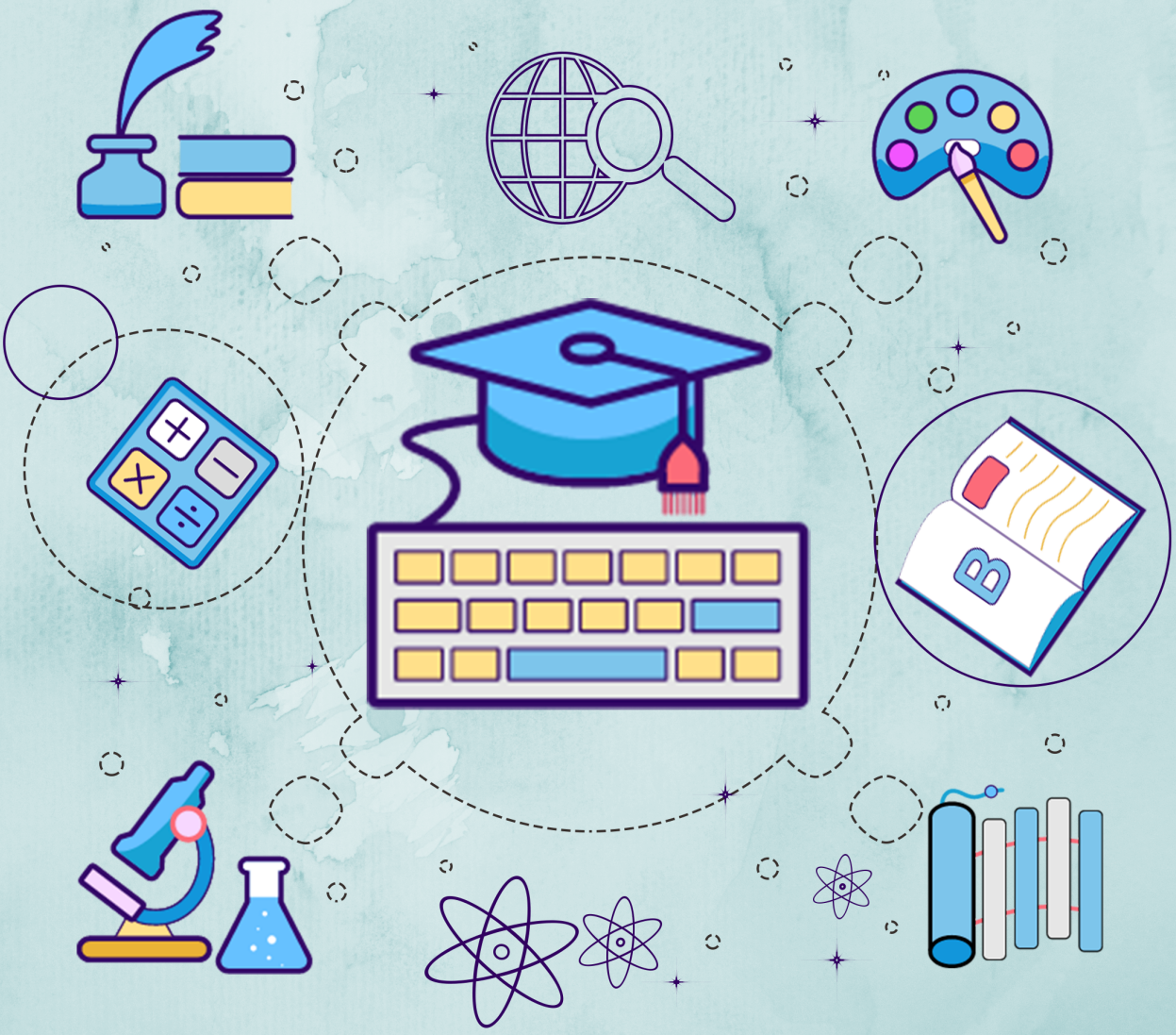
APJ ABDUL KALAM TECHNOLOGICAL UNIVERSITY

KeralaNotes



SYLLABUS | STUDY MATERIALS | TEXTBOOK

PDF | SOLVED QUESTION PAPERS

www.keralanotes.com

KeralaNotes

KTU STUDY MATERIALS

PROGRAMMING IN PYTHON

CST 362

Module 2

Related Link :

- KTU S6 CSE NOTES 2019 SCHEME
- KTU S6 SYLLABUS CSE COMPUTER SCIENCE
- KTU PREVIOUS QUESTION BANK S6 CSE SOLVED
- KTU CSE TEXTBOOKS S6 B.TECH PDF DOWNLOAD
- KTU S6 CSE NOTES | SYLLABUS | QBANK | TEXTBOOKS DOWNLOAD

Programming in Python

Module II

Functions

- A function is a block of code which only runs when it is called. You can pass data, known as parameters, into a function. A function can return data as a result.
- In Python, a function is a group of related statements that performs a specific task.
- Functions help break our program into smaller and modular chunks. As our program grows larger and larger, functions make it more organized and manageable.
- Furthermore, it avoids repetition and makes the code reusable.

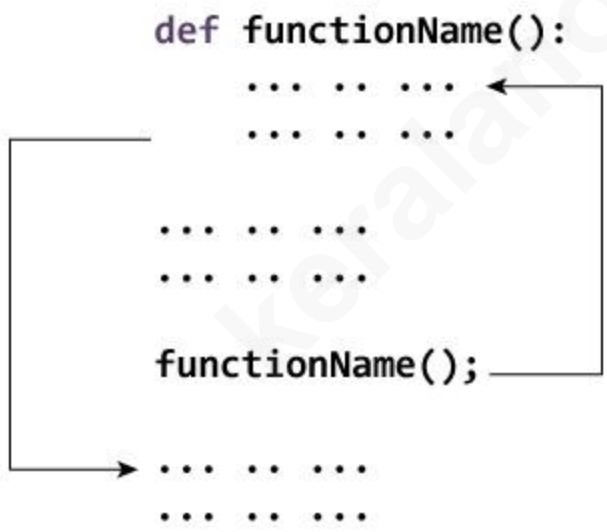
Syntax of Function

```
def function_name(parameters):
    statement(s)
```

1. Keyword ***def*** that marks the start of the function header.

2. A function name to uniquely identify the function. Function naming follows the same rules of writing identifiers in Python.
3. Parameters (arguments) through which we pass values to a function. They are optional.
4. A colon (:) to mark the end of the function header.
5. One or more valid python statements that make up the function body. Statements must have the same indentation level (usually 4 spaces).
6. An optional return statement to return a value from the function.

How Function works in Python?



Creating a Function

- In Python a function is defined using the def keyword:

Example

```
def my_function():
  print("Hello from a function")
```

Calling a Function

- To call a function, use the function name followed by parenthesis:

Example

```
def my_function():
  print("Hello from a function")
my_function()  # calling a function
```

The return statement

The return statement is used to exit a function and go back to the place from where it was called.

Syntax of return

return [expression_list]

This statement can contain an expression that gets evaluated and the value is returned. If there is no expression in the statement or the return statement itself is not present inside a function, then the function will return the None object

Example:

```
def LargestNumber(num1,num2):
    if num1 >= num2:
        return num1
    else:
        return num2
largest = LargestNumber(10,20)
print(“Largest Number is=”, largest)
print(“Largest Number is=”, LargestNumber(15,56))
```

Default Parameter Value

The following example shows how to use a default parameter value.

If we call the function without argument, it uses the default value:

Example

```
def my_function(country= "Norway"):
  print("I am from " + country)

my_function("Sweden")
my_function("India")
my_function()
my_function("Brazil")
```

Variable Scopes and Life time

- **Scope** of a variable is the portion of a program where the variable is recognized. Parameters and variables defined inside a function are not visible from outside the function. Hence, they have a local scope.
- The **lifetime** of a variable is the period throughout which the variable exits in the memory. The lifetime of variables inside a function is as long as the function executes.
- They are destroyed once we return from the function. Hence, a function does not remember the value of a variable from its previous calls.
- Here is an example to illustrate the scope of a variable inside a function.

```
def my_func():
    x = 10
    print("Value inside function:",x)

x = 20
my_func()
print("Value outside function:",x)
```

Output

```
Value inside function: 10
Value outside function: 20
```

Named arguments/ Keyword Arguments

- When we call a function with some values, these values get assigned to the arguments according to their position.
- Python allows functions to be called using keyword arguments. When we call functions in this way, the order (position) of the arguments can be changed. Following calls to the above function are all valid and produce the same result.

```
def greet(name, msg):
  print("Hello", name + ', ' + msg)


greet("Monica", "Good morning!")
greet(name = "Bruce",msg = "How do you do?")  #named argument
greet(msg = "How do you do?",name = "Bruce") #named argument
```

Main function

Main function is like the entry point of a program. However, Python interpreter runs the code right from the first line. The execution of the code starts from the starting line and goes line by line. It does not matter where the main function is present or it is present or not.

Since there is no main() function in Python, when the command to run a Python program is given to the interpreter, the code that is at level 0 indentation is to be executed. However, before doing that, it will define a few special variables. __name__ is one such special variable. If the source file is executed as the main program, the interpreter sets the __name__ variable to have a value __main__. If this file is being imported from another module, __name__ will

be set to the module's name. __name__ is a built-in variable which evaluates to the name of the current module.

Example:

```
# Python program to demonstrate  main() function
        print("Hello")
        def main():              # Defining main function
           print("hey there")
        if __name__=="__main__":          # Using the special variable __name__
           main()
```

Output:

Hello

hey there

When above program is executed, the interpreter declares the initial value of name as "main". When the interpreter reaches the if statement it checks for the value of name and when the value of if is true it runs the main function else the main function is not executed.

Main function as Module

Now when we import a Python script as module the __name__ variable gets the value same as the name of the python script imported.

Example: Let's consider there are two Files(File1.py and File2.py). File1 is as follow.

```
# File1.py

print("File1 __name__ = %s" %__name__)
```

```
if __name__ == "__main__":
        print("File1 is being run directly")
else:
        print("File1 is being imported")
```

Output:

```
File1 __name__ = __main__
File1 is being run directly
```

Now, when the File1.py is imported into File2.py, the value of __name__ changes.

```
# File2.py
import File1
print("File2 __name__ = %s" %__name__)

if __name__ == "__main__":
        print("File2 is being run directly")
else:
        print("File2 is being imported")
```

Output:

```
File1 __name__ = File1
File1 is being imported
File2 __name__ = __main__
File2 is being run directly
```

As seen above, when File1.py is run directly, the interpreter sets the __name__ variable as __main__ and when it is run through File2.py by importing, the __name__ variable is set as the name of the python script, i.e. File1. Thus, it can be said that if __name__ == "__main__" is the part of the program that runs when the script is run from the command line using a command like Python File1.py

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** to determine whether to stop or to continue with another recursive step.

Let's 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=lower+1
```

The equivalent recursive function performs similar primitive operations, but the loop is replaced with a selection statement, and the assignment statement is replaced with a **recursive call** of the function. Here is the code with these changes:

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

displayRange(4,10)
```

Lambda functions

- A lambda is an anonymous function.
- It has no name of its own, but contains the names of its arguments as well as a single expression.
- When the lambda is applied to its arguments, its expression is evaluated, and its value is returned.
- The syntax of a **lambda** is

```
lambda <argname-1, ..., argname-n>: <expression>
```

- All of the code must appear on one line and a **lambda** cannot include a selection statement, because selection statements are not expressions.
- Example:

```
x=lambda x,y:x+y
print(x(2,3))
```

output will be 5

- Example:

```
def myfunc(n):
  return lambda a : a * n
mydoubler = myfunc(2)
mytripler = myfunc(3)
print(mydoubler(11))
print(mytripler(11))
```

Strings and Number Systems

Number Systems

- **Decimal number system:** This system, also called the base ten number system, uses the ten characters 0, 1, 2, 3, 4, 5, 6, 7, 8, and 9 as digits.
- **Binary number system:** the binary number system is used to represent all information in a digital computer. The two digits in this base two number system are 0 and 1.
- **Octal number system:** This system, also called the base eight number system, uses the ten characters 0, 1, 2, 3, 4, 5, 6, and 7 as digits.
- **Hexadecimal number system:** This system, also called the base sixteen number system, uses the ten characters 0, 1, 2, 3, 4, 5, 6, 7, 8, 9, A, B, C, D, E and F as digits.

To identify the system being used, you attach the base as a subscript to the number. The digits used in each system are counted from 0 to *n* - 1, where *n* is the system's base. Thus, the digits 8 and 9 do not appear in the octal system. To represent digits with values larger than 9_{10}, systems such as base 16 use letters. Thus, A_{16} represents the quantity 10_{10}, whereas 10_{16} represents the quantity 16_{10}.

- For example, the following numbers represent the quantity 415_{10} in the binary, octal, decimal, and hexadecimal systems:

415 in binary notation	110011111_2
415 in octal notation	637_8
415 in decimal notation	415_{10}
415 in hexadecimal notation	$19F_{16}$

Handling numbers in various formats

1. **Converting Binary to Decimal:**

$1100111_2 =$

$1 * 2^6 + 1 * 2^5 + 0 * 2^4 + 0 * 2^3 + 1 * 2^2 + 1 * 2^1 + 1 * 2^0 =$

$1 * 64 + 1 * 32 + 0 * 16 + 0 * 8 + 1 * 4 + 1 * 2 + 1 * 1 =$

$64 + 32 + 4 + 2 + 1 = 103$

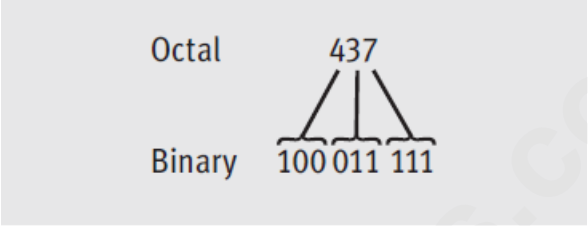
2. **Converting Decimal to Binary:**

Repeatedly divides the decimal number by 2. After each division, the remainder (either a 0 or a 1) is placed at the beginning of a string of bits. The quotient becomes the next dividend in the process. The string of bits is initially empty, and the process continues while the decimal number is greater than 0.

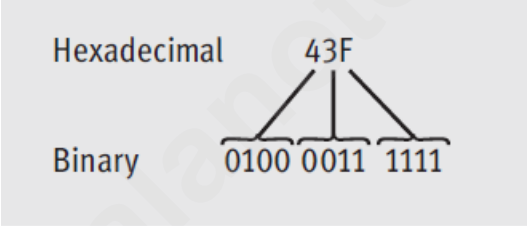
```
Enter a decimal integer: 34
Quotient Remainder Binary
   17        0          0
    8        1         10
    4        0        010
    2        0       0010
    1        0      00010
    0        1     100010
The binary representation is 100010
```

3. Octal and Hexadecimal Numbers:

a) conversion of octal to binary:



b) conversion of hexadecimal to binary



String Methods/ String function

STRING METHOD	WHAT IT DOES
`s.center(width)`	Returns a copy of `s` centered within the given number of columns.
`s.count(sub [, start [, end]])`	Returns the number of non-overlapping occurrences of substring `sub` in `s`. Optional arguments `start` and `end` are interpreted as in slice notation.
`s.endswith(sub)`	Returns `True` if `s` ends with `sub` or `False` otherwise.
`s.find(sub [, start [, end]])`	Returns the lowest index in `s` where substring `sub` is found. Optional arguments `start` and `end` are interpreted as in slice notation.
`s.isalpha()`	Returns `True` if `s` contains only letters or `False` otherwise.
`s.isdigit()`	Returns `True` if `s` contains only digits or `False` otherwise.
`s.join(sequence)`	Returns a string that is the concatenation of the strings in the sequence. The separator between elements is `s`.
`s.lower()`	Returns a copy of `s` converted to lowercase.
`s.replace(old, new [, count])`	Returns a copy of `s` with all occurrences of substring `old` replaced by `new`. If the optional argument `count` is given, only the first `count` occurrences are replaced.
`s.split([sep])`	Returns a list of the words in `s`, using `sep` as the delimiter string. If `sep` is not specified, any whitespace string is a separator.
`s.startswith(sub)`	Returns `True` if `s` starts with `sub` or `False` otherwise.
`s.strip([aString])`	Returns a copy of `s` with leading and trailing whitespace (tabs, spaces, newlines) removed. If `aString` is given, remove characters in `aString` instead.
`s.upper()`	Returns a copy of `s` converted to uppercase.

```
>>> s = "Hi there!"
>>> len(s)
9
>>> s.center(11)
' Hi there! '
>>> s.count('e')
2
>>> s.endswith("there!")
True
>>> s.startswith("Hi")
True
>>> s.find('the')
3
>>> s.isalpha()
False
>>> 'abc'.isalpha()
True
>>> "326".isdigit()
True
>>> words = s.split()
>>> words
['Hi', 'there!']
>>> "".join(words)
'Hithere!'
>>> " ".join(words)
'Hi there!'
>>> s.lower()
'hi there!'
>>> s.upper()
'HI THERE!'
>>> s.replace('i', 'o')
'Ho there!'
>>> " Hi there! ".strip()
'Hi there!'
>>>
```

Questions

1. Translate each of the following numbers to decimal numbers:
   - a 11001_2
   - b 100000_2
   - c 11111_2
2. Translate each of the following numbers to binary numbers:
   - a 47_{10}
   - b 127_{10}
   - c 64_{10}
3. Translate each of the following numbers to binary numbers:
   - a 47_8
   - b 127_8
   - c 64_8
4. Translate each of the following numbers to decimal numbers:
   - a 47_8
   - b 127_8
   - c 64_8
5. Translate each of the following numbers to decimal numbers:
   - a 47_{16}
   - b 127_{16}
   - c AA_{16}

- A **data structure** combines several data values into a unit so they can be treated as one thing.
- The data elements within a data structure are usually organized in a special way that allows the programmer to access and manipulate them.
- A **string** is a data structure that organizes text as a sequence of characters.
- In this chapter, we explore the use of two other common data structures:
  - **List**
  - **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.

Lists

- A list is a sequence of data values called **items or elements**. An item can be of any type.
- Each of the items in a list is ordered by position.
- Each item in a list has a unique **index** that specifies its position.
- The index of the first item is 0, and the index of the last item is the length of the

list minus 1.

- In Python, a list is written as a sequence of data values separated by commas. The entire sequence is enclosed in square brackets (**[** and **]**).

```
[1951, 1969, 1984]                    # A list of integers

['apples', 'oranges', 'cherries']     # A list of strings

[]                                    # An empty list
```

- You can also use other lists as elements in a list, thereby creating a list of lists.

```
[[5, 9], [541, 78]]
```

- When the element is a variable or any other expression, its value is included in the list.

```
>>> import math
>>> x = 2
>>> [x, math.sqrt(x)]
[2, 1.4142135623730951]
>>> [x + 1]
[3]
>>>
```

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

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

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

```
>>> len(first)
4
>>> first[0]
1
>>> first[2:4]
[3, 4]
>>>
```

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

```
>>> first + [5, 6]
[1, 2, 3, 4, 5, 6]
>>> first == second
True
>>>
```

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

```
>>> print("1234")
1234
>>> print([1, 2, 3, 4])
[1, 2, 3, 4]
>>>
```

- To print the contents of a list without the brackets and commas, you can use a **for** loop, as follows:

```
>>> for element in [1, 2, 3, 4]:
        print(element, end=" ")

1 2 3 4
>>>
```

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

```
>>> 3 in [1, 2, 3]
True
>>> 0 in [1, 2, 3]
False
>>>
```

- Table summarizes these operators and functions, where **L** refers to a list.

OPERATOR OR FUNCTION	WHAT IT DOES
`L[<an integer expression>]`	Subscript used to access an element at the given index position.
`L[<start>:<end>]`	Slices for a sublist. Returns a new list.
`L + L`	List concatenation. Returns a new list consisting of the elements of the two operands.
`print(L)`	Prints the literal representation of the list.
`len(L)`	Returns the number of elements in the list.
`list(range(<upper>))`	Returns a list containing the integers in the range `0` through `upper - 1`.
`==, !=, <, >, <=, >=`	Compares the elements at the corresponding positions in the operand lists. Returns `True` if all the results are true, or `False` otherwise.
`for <variable> in L:` `  <statement>`	Iterates through the list, binding the 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

- The subscript operator is used to replace an element at a given position, as shown in the next session:

```
>>> example = [1, 2, 3, 4]
>>> example
[1, 2, 3, 4]
>>> example[3] = 0
>>> example
[1, 2, 3, 0]
>>>
```

- Much of list processing involves replacing each element, with the result of

applying some operation to that element.

```
>>> numbers = [2, 3, 4, 5]
>>> numbers
[2, 3, 4, 5]
>>> index = 0
>>> while index < len(numbers):
        numbers[index] = numbers[index] ** 2
        index += 1

>>> numbers
[4, 9, 16, 25]
>>>
```

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

```
>>> numbers = list(range(6))
>>> numbers
[0, 1, 2, 3, 4, 5]
>>> numbers[0:3] = [11, 12, 13]
>>> numbers
[11, 12, 13, 3, 4, 5]
>>>
```

List Methods for Inserting and Removing Elements

LIST METHOD	WHAT IT DOES
`L.append(element)`	Adds `element` to the end of `L`.
`L.extend(aList)`	Adds the elements of `aList` to the end of `L`.
`L.insert(index, element)`	Inserts `element` at `index` if `index` is less than the length of `L`. Otherwise, inserts `element` at the end of `L`.
`L.pop()`	Removes and returns the element at the end of `L`.
`L.pop(index)`	Removes and returns the element at `index`.

```
>>> example = [1, 2]
>>> example
[1, 2]
>>> example.insert(1, 10)
>>> example
[1, 10, 2]
>>> example.insert(3, 25)
>>> example
[1, 10, 2, 25]
>>>
```

```
>>> example = [1, 2]
>>> example
[1, 2]
>>> example.append(10)
>>> example
[1, 2, 10]
>>> example.extend([11, 12, 13])
>>> example
[1, 2, 10, 11, 12, 13]
>>>
```

```
>>> example
[1, 2, 10, 11, 12, 13]
>>> example.pop()
13
>>> example
[1, 2, 10, 11, 12]
>>> example.pop(0)
1
>>> example
[2, 10, 11, 12]
>>>
```

Searching a List

- After elements have been added to a list, a program can search for a given element.
- The ***in*** operator determines an element's presence or absence, but programmers often are more interested in the position of an element if it is found (for replacement, removal, or other use). Unfortunately, the **list** type does not include the convenient ***find*** method that is used with strings.
- Instead of **find**, you must use the method **index** to locate an element's position in a list.
- It is unfortunate that **index** raises an error when the target element is not found.
- To guard against this unpleasant consequence, you must first use the **in** operator to test for presence and then the **index** method if this test returns **True**.

```
aList = [34, 45, 67]
target = 45
if target in aList:
    print(aList.index(target))
else:
    print(-1)
```

Sorting a List

- Although a list's elements are always ordered by position, it is possible to impose a natural ordering on them as well.
- In other words, you can arrange some elements in numeric or alphabetical order. A list of numbers in ascending order and a list of names in alphabetical order are sorted lists.
- When the elements can be related by comparing them for less than and greater than as well as equality, they can be sorted.
- The list method *sort* mutates a list by arranging its elements in ascending order. Here is an example of its use:

```
>>> example = [4, 2, 10, 8]
>>> example
[4, 2, 10, 8]
>>> example.sort()
>>> example
[2, 4, 8, 10]
```

List comprehension

- List comprehensions are used for creating new lists from other iterables.
- As list comprehensions return lists, they consist of brackets containing the expression, which is executed for each element along with the for loop to iterate over each element.
- This is the basic syntax:

new_list = [expression for_loop_one_or_more conditions]

- For example, find the squares of a number using the for loop:

```
numbers=[1,2,3,4]
squares=[]
    for n in numbers:
        squares.append(n**2)
print(squares)  # Output: [1, 4, 9, 16]
```

- Finding squares using list comprehensions:

```
numbers = [1, 2, 3, 4]
squares = [n**2 for n in numbers]
print(squares)  # Output: [1, 4, 9, 16]
```

- Find common numbers from two lists using list comprehension:???????????

Tuples

- A tuple is a type of sequence that resembles a list, except that, unlike a list, a tuple is immutable. You indicate a tuple literal in Python by enclosing its elements in parentheses instead of square brackets. The next session shows how to create several tuples:

```
>>> fruits = ("apple", "banana")
>>> fruits
('apple', 'banana')
>>> meats = ("fish", "poultry")
>>> meats
('fish', 'poultry')
>>> food = meats + fruits
>>> food
('fish', 'poultry', 'apple', 'banana')
>>> veggies = ["celery", "beans"]
>>> tuple(veggies)
('celery', 'beans')
```

- You can use most of the operators and functions used with lists in a similar fashion with tuples.

Dictionaries

- Lists organize their elements by position.
- However, in some situations, the position of a datum in a structure is irrelevant;

we're interested in its association with some other element in the structure.

- A dictionary organizes information by **association**, not position.
- For example, when you use a dictionary to look up the definition of "mammal," you don't start at page 1; instead, you turn directly to the words beginning with "M."
- In Python, a **dictionary** associates a set of **keys** with data values.
- A Python dictionary is written as a sequence of key/value pairs separated by commas.
- These pairs are sometimes called entries. The entire sequence of entries is enclosed in curly braces ({ and }). A colon (:) separates a key and its value.
- Here are some example dictionaries:

  A phone book: {'Savannah':'476-3321', 'Nathaniel':'351-7743'}

  Personal information: {'Name':'Molly', 'Age':18}

- You can even create an empty dictionary—that is, a dictionary that contains no entries.

```
{}
```

Adding Keys and Replacing Values

- You add a new key/value pair to a dictionary by using the subscript operator [].
- The form of this operation is the following:

```
<a dictionary>[<a key>] = <a value>
```

- The next code segment creates an empty dictionary and adds two new entries:

```
>>> info = {}
>>> info["name"] = "Sandy"
>>> info["occupation"] = "hacker"
>>> info
{'name': 'Sandy', 'occupation': 'hacker'}
>>>
```

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

```
>>> info["occupation"] = "manager"
>>> info
{'name': 'Sandy', 'occupation': 'manager'}
>>>
```

Accessing Values

- You can also use the subscript to obtain the value associated with a key. However, if the key is not present in the dictionary, Python raises an error. Here are some examples, using the info dictionary, which was set up earlier:

```
>>> info["name"]
'Sandy'
>>> info["job"]
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
KeyError: 'job'
>>>
```

- If the existence of a key is uncertain, the programmer can test for it using the dictionary method **has_key**, but a far easier strategy is to use the method **get**. This method expects two arguments, a possible key and a default value. If the key is in the dictionary, the associated value is returned. However, if the key is absent, the default value passed to **get** is returned. (has_key was removed in python3)

- Here is an example of the use of **get** with a default value of **None**:

```
>>> print(info.get("job", None))
None
>>>
```

Removing Keys

- To delete an entry from a dictionary, one removes 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 is used with just one argument, and this key is absent from the dictionary, Python raises an error.
- The next session attempts to remove two keys and prints the values returned:

```
>>> print(info.pop("job", None))
None
>>> print(info.pop("occupation"))
manager
>>> info
{'name': 'Sandy'}
>>>
```

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.
- The next code segment prints all of the keys and their values in our info dictionary:

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

- Alternatively, you could use the dictionary method **items()** to access a list of the dictionary's entries. The next session shows a run of this method with a dictionary of grades:

```
>>> grades = {90:"A", 80:"B", 70:"C"}
>>> grades.items()
[(80, 'B'), (90, 'A'), (70, 'C')]
```

- Note that the entries 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:

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

- If a special ordering of the keys is needed, you can obtain a list of keys using the **keys** method and process this list to rearrange the keys.
- For example, you can sort the list and then traverse it to print the entries of the dictionary in alphabetical order:

```
theKeys = list(info.keys())
theKeys.sort()
for key in theKeys:
    print(key, info[key])
```

- Table below summarizes the commonly used dictionary operations, where **d** refers to a dictionary.

DICTIONARY OPERATION	WHAT IT DOES
`len(d)`	Returns the number of entries in **d**.
`aDict[key]`	Used for inserting a new key, replacing a value, or obtaining a value at an existing key.
`d.get(key [, default])`	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.
`d.pop(key [, default])`	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.
`list(d.keys())`	Returns a list of the keys.
`list(d.values())`	Returns a list of the values.

`list(d.items())`	Returns a list of tuples containing the keys and values for each entry.
`d.has_key(key)`	Returns `True` if the key exists or `False` otherwise.
`d.clear()`	Removes all the keys.
`for key in d:`	`key` is bound to each key in `d` in an unspecified order.

Set

- A set is a collection which is unordered and unindexed. In Python, sets are written with curly brackets.
- **Create a Set:**

```
thisset = {"apple", "banana", "cherry"}
print(thisset)
```

- Sets are unordered, so you cannot be sure in which order the items will appear.

- **Access Items:** You cannot access items in a set by referring to an index or a key.
- But you can loop through the set items using a *for loop*, or ask if a specified value is present in a set, by using the in keyword.

```
thisset = {"apple", "banana", "cherry"}
for x in thisset:
  print(x)
```

- Check if "banana" is present in the set:

```
thisset = {"apple", "banana", "cherry"}
print("banana" in thisset)
```

- **Add Items:** To add one item to a set use the add() method. To add more than one item to a set use the update() method.

#using the add() method

```
thisset = {"apple", "banana", "cherry"}

thisset.add("orange")

print(thisset)
```

#using the update() method

```
thisset = {"apple", "banana", "cherry"}

thisset.update(["orange", "mango", "grapes"])

print(thisset)
```

- **Get the Length of a Set:** To determine how many items a set has, use the len() method.

```
thisset = {"apple", "banana", "cherry"}
print(len(thisset))
```

- **Remove Item:** To remove an item in a set, use the remove(), or the discard() method. If the item to remove does not exist, remove() will raise an error. If the item to remove does not exist, discard() will NOT raise an error. You can also use the pop(), method to remove an item, but this method will remove the *last* item. The return value of the pop() method is the removed item.

```
#using the remove() method

    thisset = {"apple", "banana", "cherry"}
    thisset.remove("banana")
    print(thisset)

#using the discard() method

    thisset = {"apple", "banana", "cherry"}
    thisset.discard("banana")
    print(thisset)

#using the pop() method

    thisset = {"apple", "banana", "cherry"}
    x = thisset.pop()
    print(x)
    print(thisset)
```

Python Sets Methods	
Method	**Description**
add()	Adds an element to the set
clear()	Removes all the elements from the set
copy()	Returns a copy of the set
difference()	Returns a set containing the difference between two or more sets
difference_update()	Removes the items in this set that are also included in another, specified set
discard()	Remove the specified item
intersection()	Returns a set, that is the intersection of two other sets
intersection_update()	Removes the items in this set that are not present in other, specified set(s)
isdisjoint()	Returns whether two sets have a intersection or not
issubset()	Returns whether another set contains this set or not
issuperset()	Returns whether this set contains another set or not
pop()	Removes an element from the set
remove()	Removes the specified element
symmetric_difference()	Returns a set with the symmetric differences of two sets
symmetric_difference_update()	inserts the symmetric differences from this set and another
union()	Return a set containing the union of sets
update()	Update the set with the union of this set and others

Datetime

- **Dates:** A date in Python is not a data type of its own, but we can import a module named ***datetime*** to work with dates as date objects.
- Import the datetime module and display the current date:

```
import datetime
x = datetime.datetime.now()
print(x)
```

- **Creating Date Objects:** To create a date, we can use the datetime() class (constructor) of the datetime module. The datetime() class requires three parameters to create a date: year, month, day.

```
import datetime
x = datetime.datetime(2020, 5, 17)
print(x)
```

- The datetime() class also takes parameters for time and timezone (hour, minute, second, microsecond, tzone), but they are optional, and has a default value of 0, (None for timezone).

- **The strftime() Method**: The datetime object has a method for formatting date objects into readable strings. The method is called strftime(), and takes one parameter, ***format,*** to specify the format of the returned string:

```
import datetime
x = datetime.datetime(2018, 6, 1)
print(x.strftime("%B"))
```

- A reference of all the legal format codes:

Format	Meaning	Example
%a	abbreviated weekday name	Mon
%A	full weekday name	Monday
%b	abbreviated month name	Dec
%B	full month name	December
%c	the locale's date and time	Sun Oct 22 00:00:00 2019
%d	zero-padded day of the month	[01,31]
%e	number [1,31]; equivalent to %_d.	[1,31]
%H	hour in 24-hour clock	[00,23]
%I	hour in 12-hour clock	[01,12]
%j	day of the year	[001,366]
%m	month as a number	[01,12]
%M	minute as a number	[00,59]
%S	seconds as a number	[000, 999]
%p	either AM or PM	PM
%U	Sunday-based week of the year	[00,53]
%w	Sunday-based weekday	[0,6]
%W	Monday-based week of the year	[00,53]
%x	the locale's date as %-m/%-d/%Y	07/17/2019
%X	the locale's time as %-I:%M:%S %p	11:24:45
%y	year without century	[00,99]
%Y	year with century	2019
%Z	time zone offset	-0700, -07:00, -07, or Z
%%	a literal percent sign	%