UNIT – III

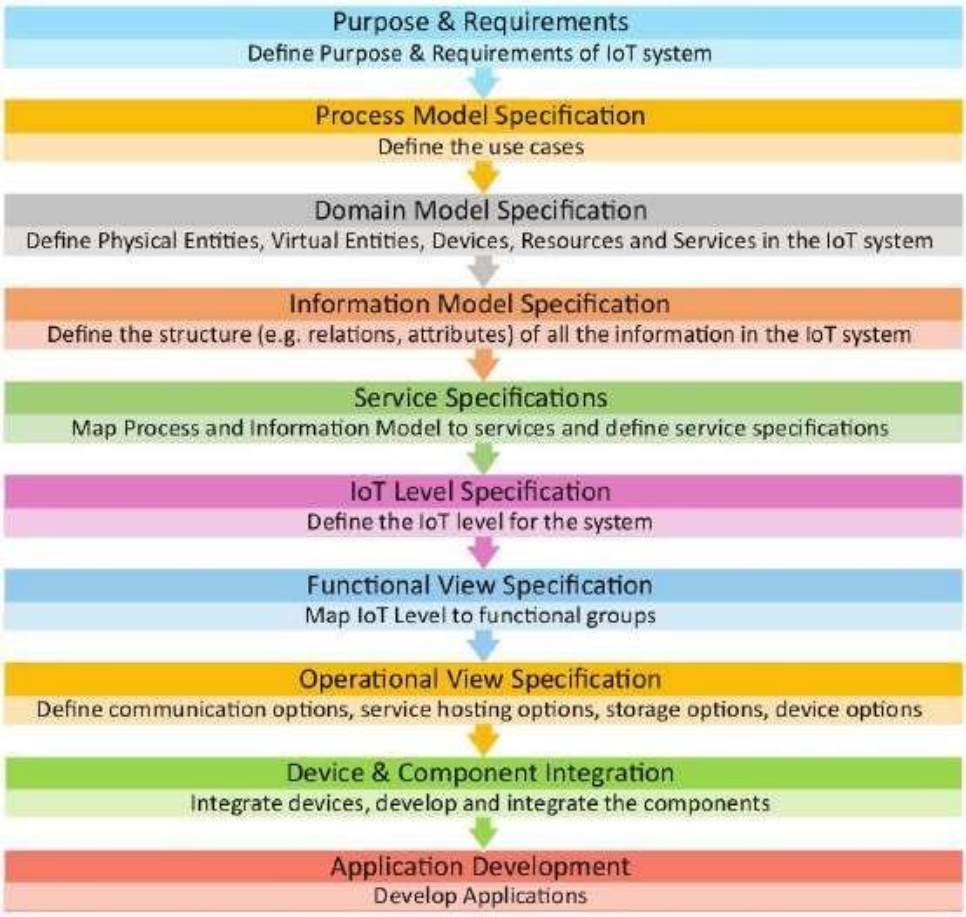
IoT Platforms Design Methodology: IoT Design Methodology, Case Study on IoT System for Weather Monitoring, Motivation for Using Python, IoT Systems - Logical Design using Python, Installing Python, Python Data Types & Data Structures, Control Flow, Functions, Modules, Packages, File Handling, Date/Time Operations, Classes, Python Packages of Interest for IoT.

IOT PLATFORMS DESIGN METHODOLOGY

IoT Design Methodology:

IoT Design Methodology includes:

1. Purpose & Requirements Specification
2. Process Specification
3. Domain Model Specification
4. Information Model Specification
5. Service Specifications
6. IoT Level Specification
7. Functional View Specification
8. Operational View Specification
9. Device & Component Integration
10. Application Development



IoT Design Methodology – Steps

Step 1: Purpose & Requirements Specification:

The first step in IoT system design methodology is to define the purpose and requirements of the system. In this step, the system purpose, behavior and requirements (such as data collection requirements, data analysis requirements, system management requirements, data privacy and security requirements, user interface requirements, ...) are captured.

Applying this to the example of Smart Home Automation System, the purpose and requirements for the system may be described as follows:

Purpose: A home automation system that allows controlling of the lights in a home remotely using a web application.

Behavior: The home automation system should have auto and manual models. In auto mode, the system measures light level in the room and switches on the light when it gets dark. In manual mode, the system provides the option of manually and remotely switching on/off the light.

System Management Requirement: The system should provide remote monitoring and control functions.

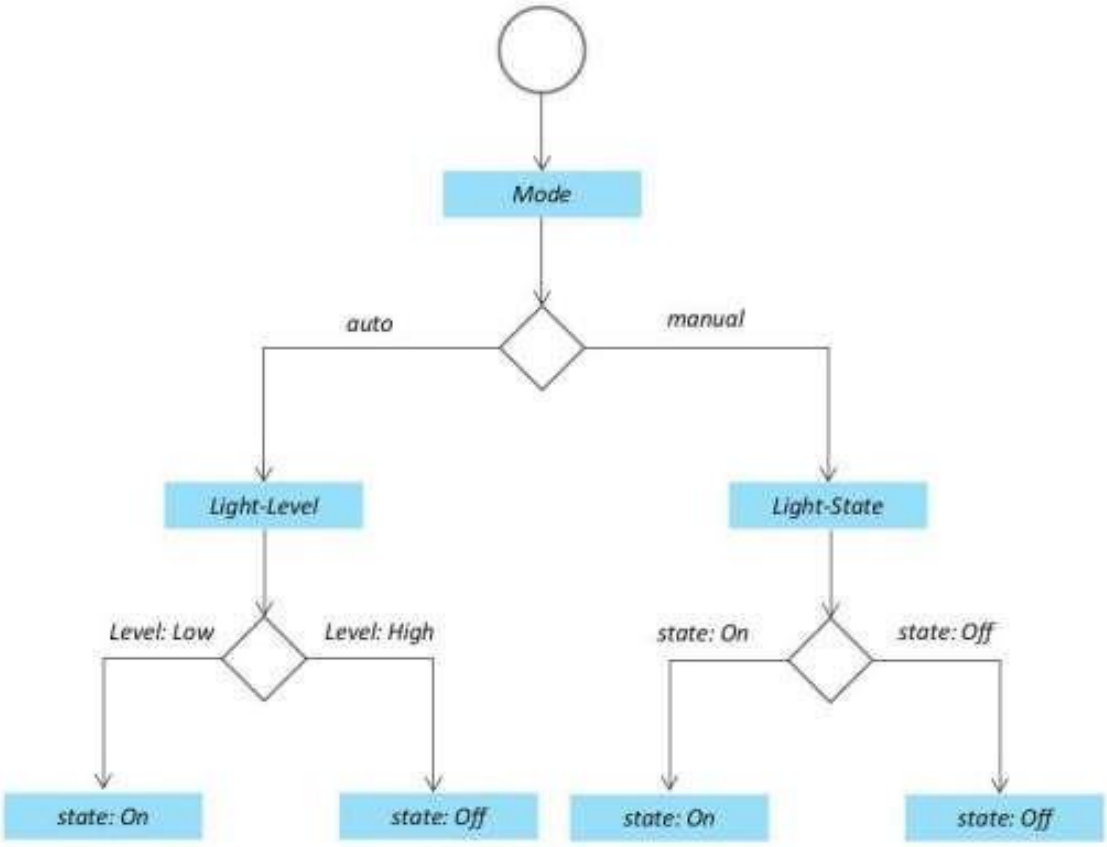
Data Analysis Requirement: The system should perform local analysis of the data.

Application Deployment Requirement: The application should be deployed locally on the device, but should be accessible remotely.

Security Requirement: The system should have basic user authentication capability.

Step 2: Process Specification:

The second step in the IoT design methodology is to define the process specification. In this step, the use cases of the IoT system are formally described based on and derived from the purpose and requirement specifications. The following diagram shows the process diagram for the home automation.

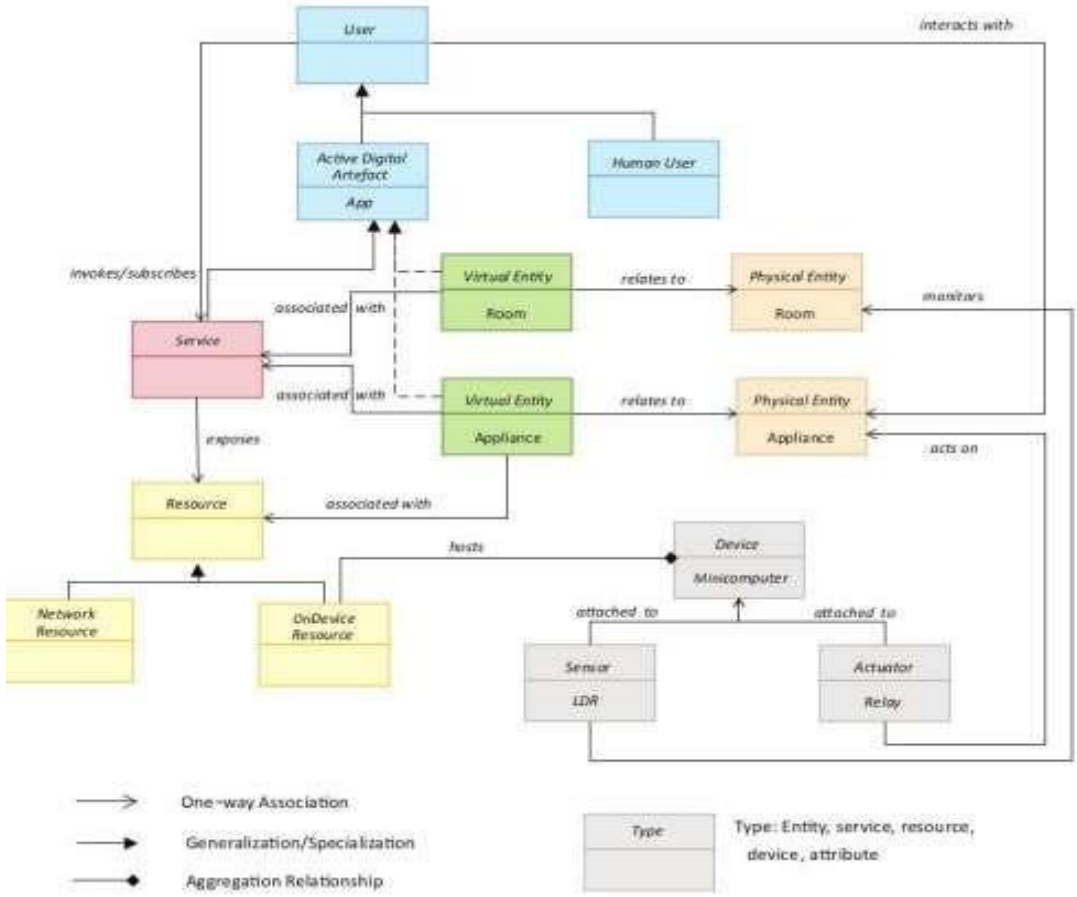


The process diagram shows the two modes of the system – auto and manual.

In the above figure, the circle denotes the start of a process, diamond denotes a decision box and rectangle denotes a state or attribute. When auto mode is chosen, the system monitors the light level. If the level is low, the system changes the state of the light to "on". Whereas, if the light level is high, the system changes the state of the light to "off". When the manual mode is chosen, the system checks the light state set by the user. If the light state set by the user is "on", the system changes the state of light to "on". Whereas, if the state set by the user is "off", the system changes the state of light to "off".

Step 3: Domain Model Specification:

The third step in the IoT design methodology is to define the Domain Model. The domain model describes the main concepts, entities and objects in the domain of IoT system to be designed. Domain model defines the attributes of the objects and relationships between objects. Domain model provides an abstract representation of the concepts, objects and entities in the IoT domain, independent of any specific technology or platform. With the domain model, the IoT system designers can get an understanding of the IoT domain for which the system is to be designed. The following figure shows the domain model for the home automation system example. The entities, objects and concepts defined in the domain model include:



a) Physical Entity: This is a discrete and identifiable entity in the physical environment (e.g., a room, a light, a car etc.). The IOT system provides information about the Physical entity (using sensors) or performs actuation upon the Physical entity. In the home automation example, there are two physical entities involved – one is the room in the home (of which the lighting conditions are to be monitored) and the other is the light appliance to be controlled.

b) Virtual Entity: This is a representation of the Physical entity in the digital world. For each Physical entity, there is Virtual entity in the domain model. In the home automation

example, there is one Virtual entity for the room to be monitored, another for the appliance to be controlled.

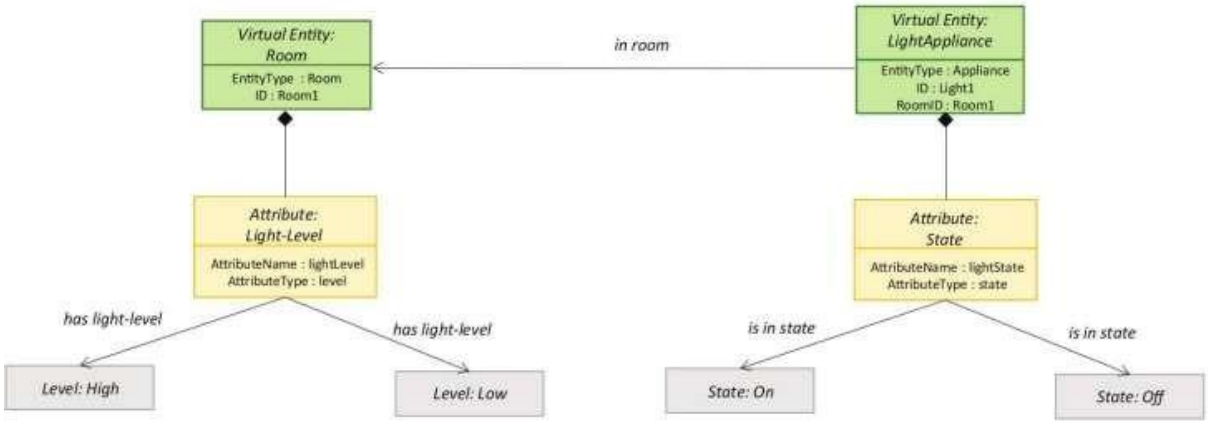
c) Device: This provides a medium for interactions between physical entities and Virtual Entities. Devices are either attached to Physical entities or placed near Physical entities. Devices are used to gather information about Physical entities (e.g., sensors), perform actuation upon Physical entities (e.g., actuators) or used to identify Physical entities (e.g., using tags). In the home automation example, the device is a single-board minicomputer which has light sensor and actuator (relay switch) attached to it.

d) Resource: Resources are software components which can be either "on-device" or "network-resources". On-device resources are hosted on the device and include software components that either provide information on or enable actuation upon the Physical entity to which the device is attached. Network resources include the software components that are available in network (such as a database). In the home automation example, the on-device resource is the operating system that runs on the single-board mini-computer.

e) Service: Services provide an interface for interacting with the Physical entity. Services access the resources hosted on the device or the network resources to obtain information about the Physical entity or perform actuation upon the Physical entity. In the home automation example, there are three services: i) a service that sets mode to auto or manual, retrieves the current mode. ii) a service that sets the light appliance sate to on/off, or retrieves the current light state. iii) a controller service that runs as a native service on the device. When in auto mode, the controller service monitors the light level and switches the light on/off and updates the status in the status database. When in manual mode, the controller service retrieves the current state from the database and switches the light from on/off.

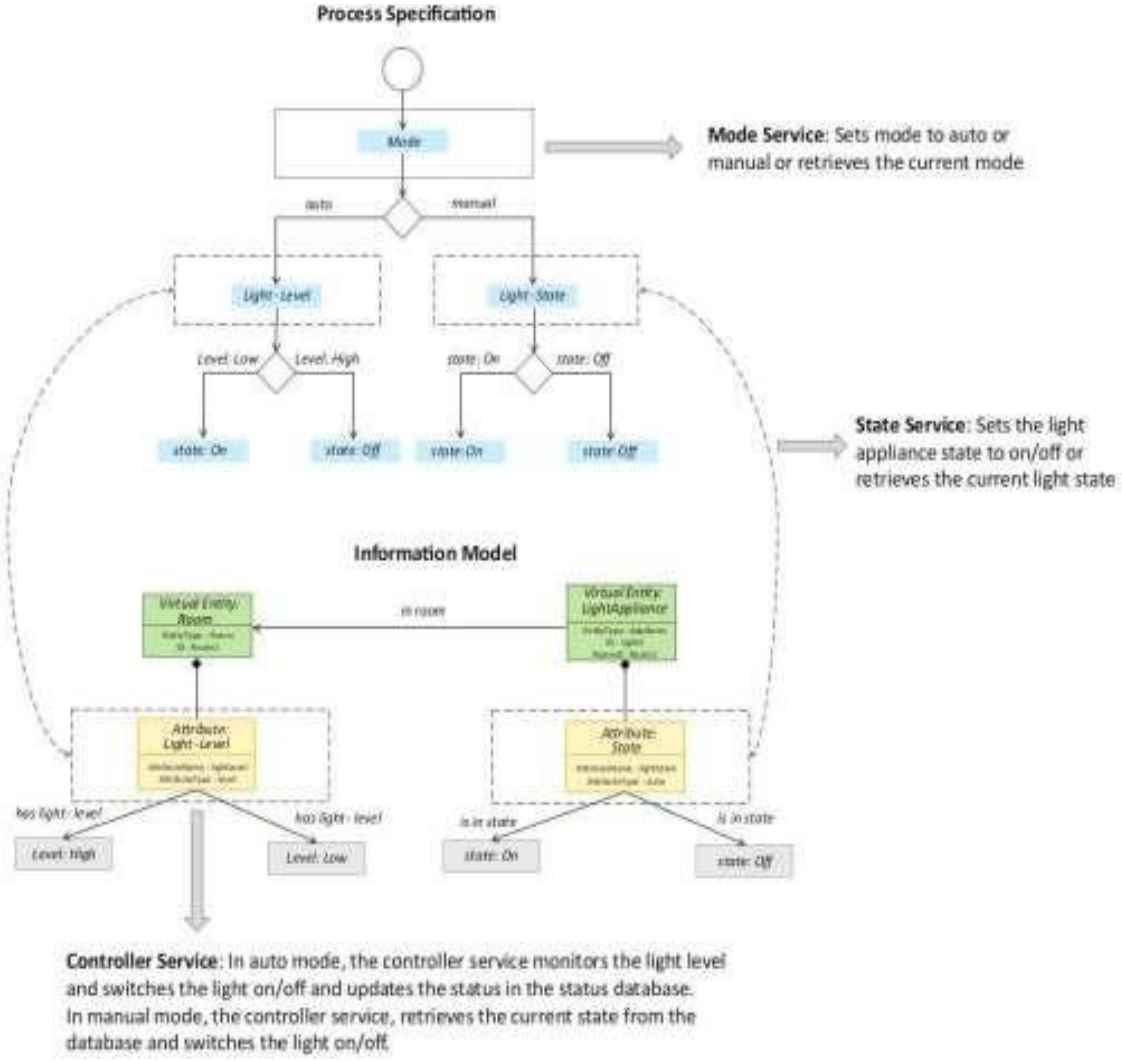
Step 4: Information Model Specification:

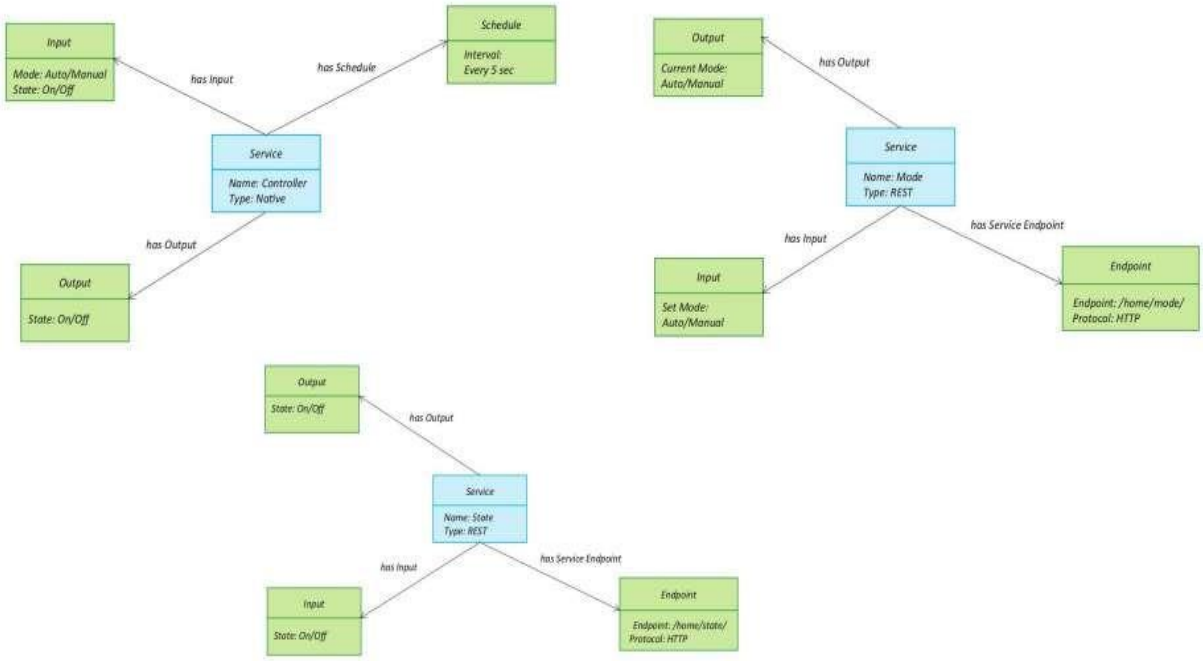
The fourth step in the IoT design methodology is to define the Information Model. Information Model defines the structure of all the information in the IoT system, for example, attributes of Virtual Entities, relations, etc. Information model does not describe the specifics of how the information is represented or stored. To define the information model, we first list the Virtual Entities defined in the Domain Model. Information model adds more details to the Virtual Entities by defining their attributes and relations. The following figure show the Information Model for the home automation example.



Step 5: Service Specifications:

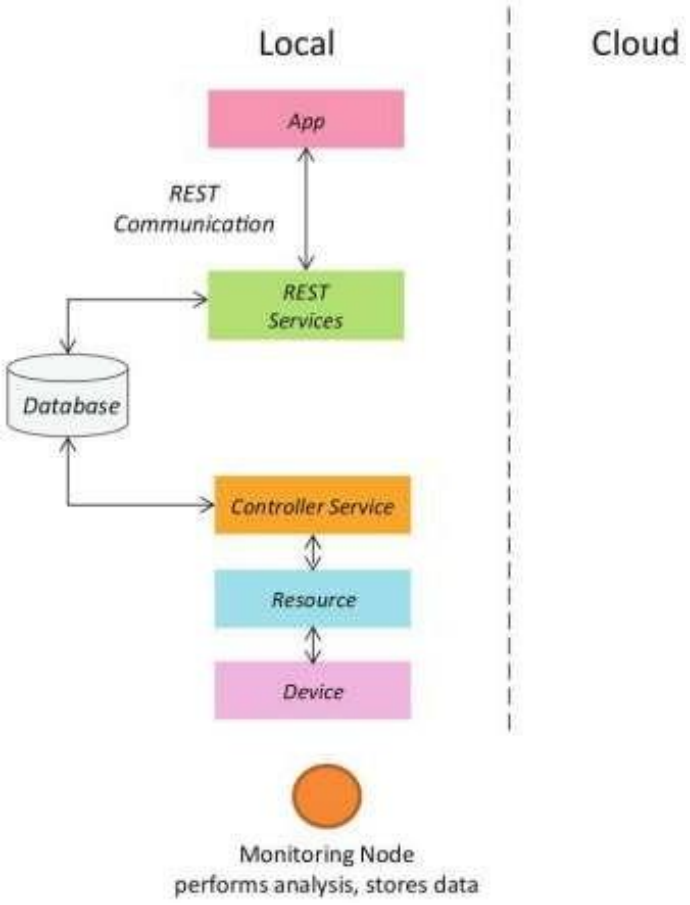
The fifth step in the IoT design methodology is to define the service specifications. Service specifications define the services in the IoT system, service types, service inputs/output, service endpoints, service schedules, service preconditions and service effects. The following figure shows an example of deriving the services from the process specification and information model, we identify the states and attributes.





Step 6: IoT Level Specification:

The sixth step in the IoT design methodology is to define the IoT level for the system. In Chapter-1, we defined five IoT deployment levels. The following figure shows the deployment level of the home automation IoT, which is level-1.



Step 7: Functional View Specification:

The seventh step in the IoT design methodology is to define the Functional View. The Functional View (FV) defines the functions of the IoT systems grouped into various Functional Groups (FGs). Each Functional Group either provides functionalities for interacting with instances of concepts defined in the Domain Model or provides information related to these concepts. The Functional Groups (FGs) included in a Functional View include:

a) Device: The device FG contains devices for monitoring and control. In the home automation example, the device FG includes a single board mini-computer, a light, sensor and relay switch (actuator).

b) Communication: The communication FG handles the communication for IoT system. The communication FG includes the communication protocols that form the backbone of IoT systems and enable network connectivity.

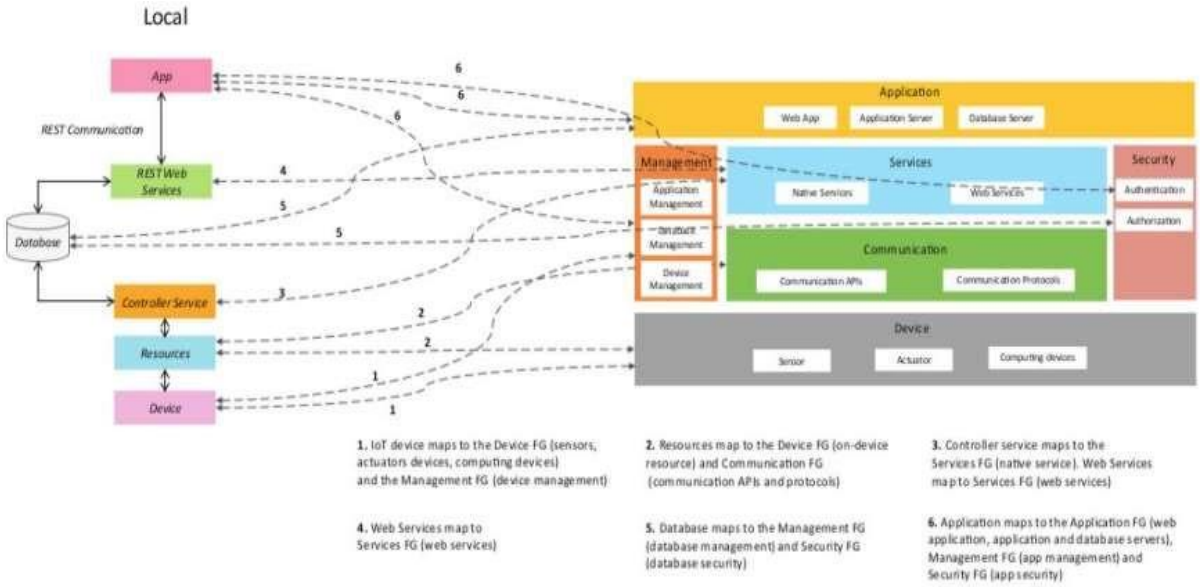
c) Services: The service FG includes various services involved in the IoT system such as services for device monitoring, device control services, data publishing services and services for device discovery. In home automation example, there are two REST services (mode and state service) and one native service (controller service).

d) Management: The management FG includes all functionalities that are needed to configure and manage the IoT system.

e) Security: The security FG includes security mechanisms for the IoT system such as authentication, authorization, data security, etc.

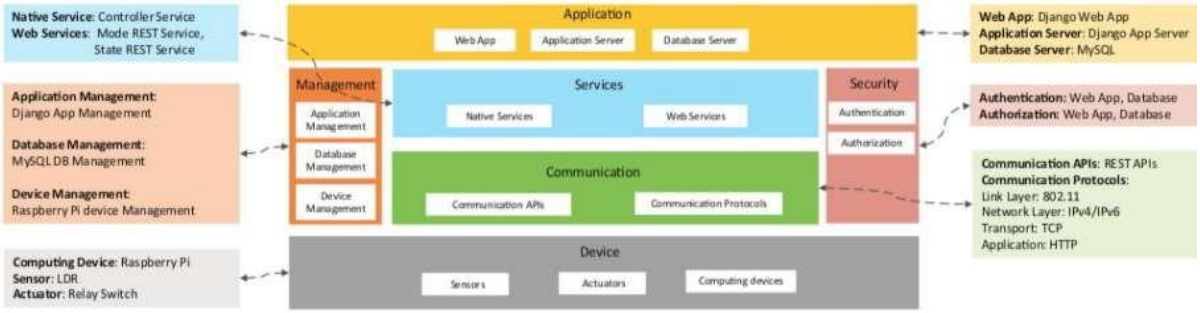
f) Application: The application FG includes applications that provide an interface to the users to control and monitor various aspects of the IoT system. Applications also allow users to view the system status and the processed data.

The following figure shows an example of mapping deployment level to functional groups for home automation IOT system.



Step 8: Operational View Specification:

The eighth step in the IoT design methodology is to define the Operational View Specifications. In this step, various options pertaining to the IoT system deployment and operation are defined, such as; service hosting options, storage options, device options, application hosting options, etc. The following figure shows an example of mapping functional groups to operational view specifications for home automation IoT system.



a) Devices: Computing device (Rsberry Pi), light dependent resistor (sensor), relay switch (actuator).

b) Communication APIs: REST APIs.

c) Communication Protocols: Link Layer – 802.11, Network Layer – IPV4/IPV6, Transport – TCP, Application – HTTP.

d) Services:

i. Controller Service – Hosted on device, implemented in Python and run as native service.

ii. Mode Service – REST – ful web service, hosted on device, implemented with Django-REST Framework.

iii. State Service – REST – ful web service, hosted on device, implemented with Dajango-REST framework.

e) Application:

WEB Application – Django Web Application,

Application Server – Django App Server,

Database Server – MySQL.

f) Security:

Authentication: Web App, Database

Authorization: Web App, Database

g) Management:

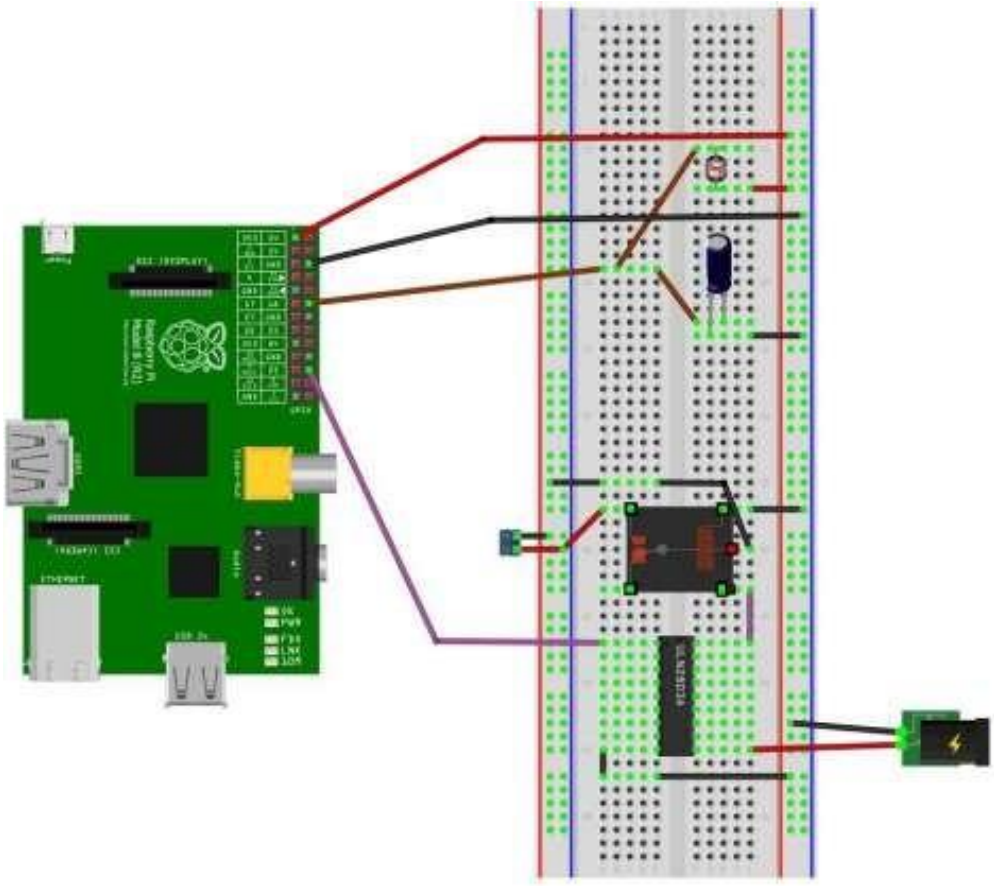
Application Management – Django App Management

Database Management – MySQL DB Management

Device Management – Raspberry Pi device Management

Step 9: Device & Component Integration:

The ninth step in the IoT design methodology is the integration of the devices and components. The following diagram shows a schematic diagram of the home automation IoT system showing the device, sensor and actuator integrated.

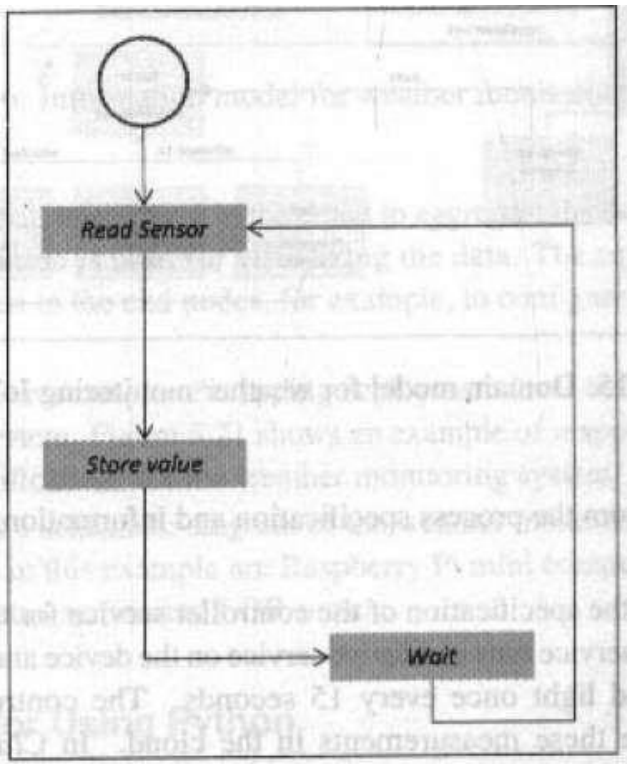


Step 10: Application Development:

The final step in the IoT design methodology is to develop the IoT application. The following figure shows the home automation web application.

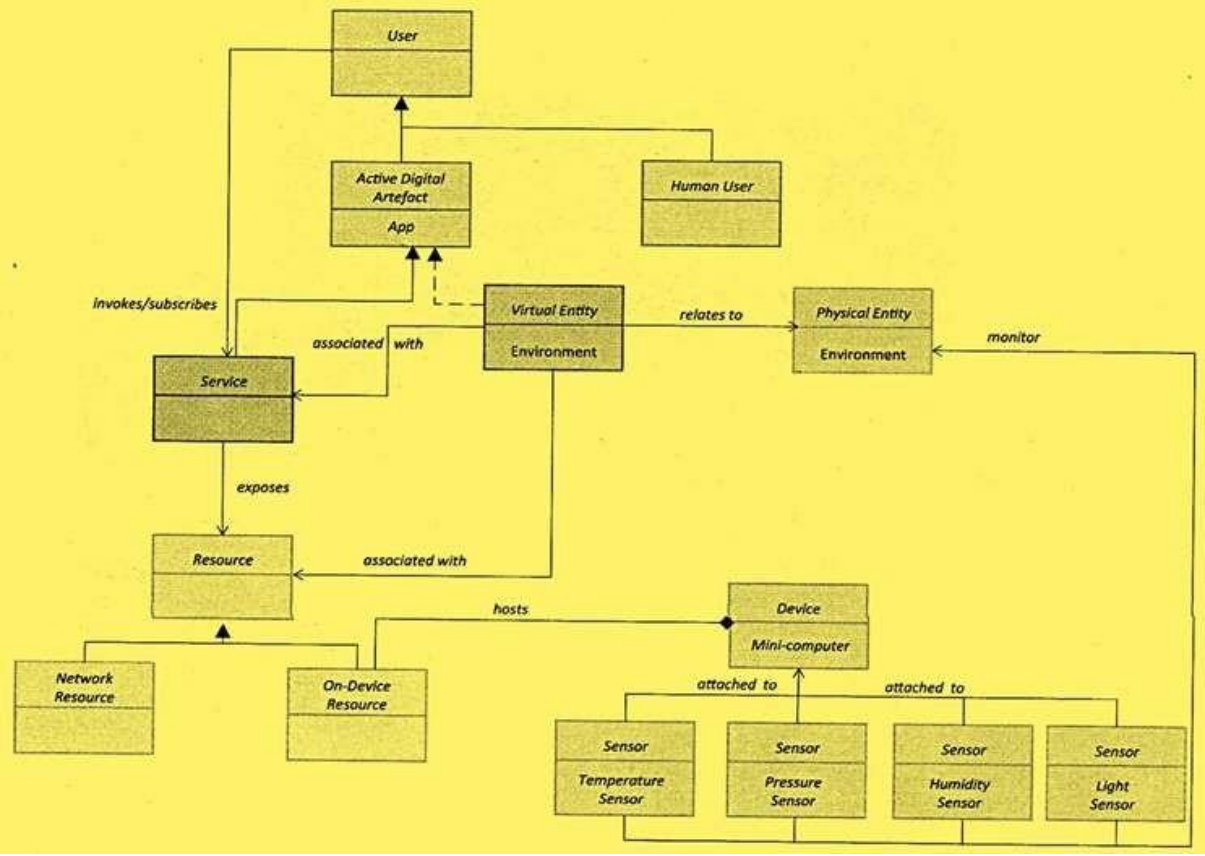
Case Study on IoT System for Weather Monitoring:

The purpose of the whether monitoring system is to collect data on environmental conditions such as temperature, pressure, humidity and light in an area using multiple end nodes. The end nodes send the data cloud where the data is aggregated and analyzed. The following figure shows the process specification for the whether monitoring system. The process specification shows that the sensors are read after fixed intervals and the sensor measurements are stored.

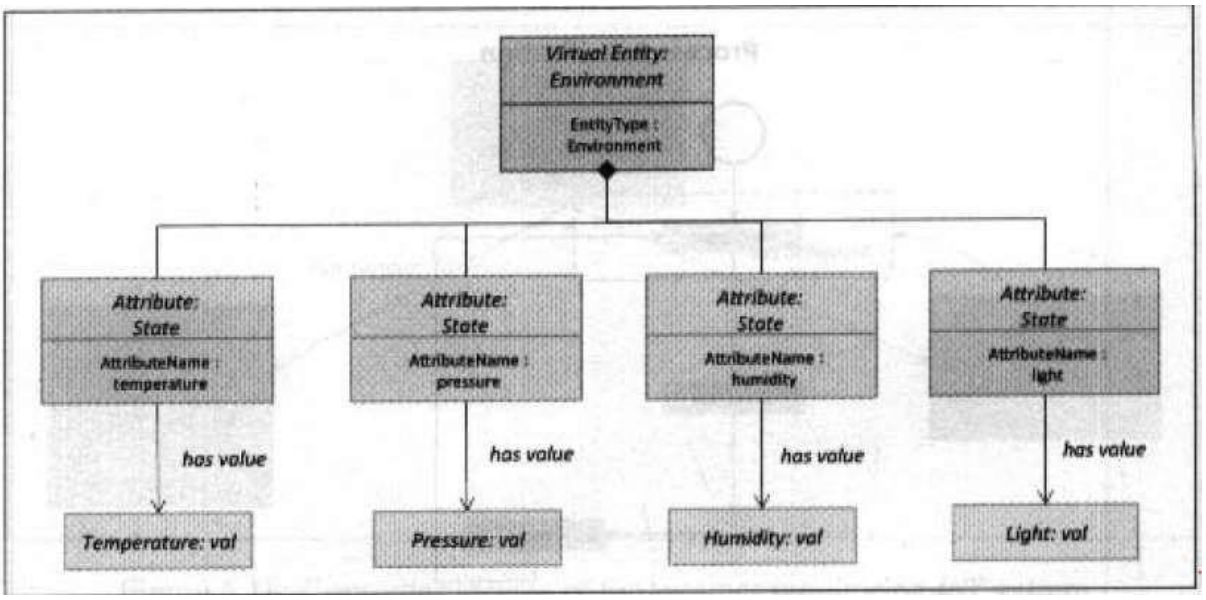


The following figure shows the domain model for the whether monitoring system. In this domain model the physical entity is the environment which is being monitored. There is a virtual entity for the environment. Devices include temperature sensor, pressure sensor, humidity sensor, light sensor and single-board minicomputer.

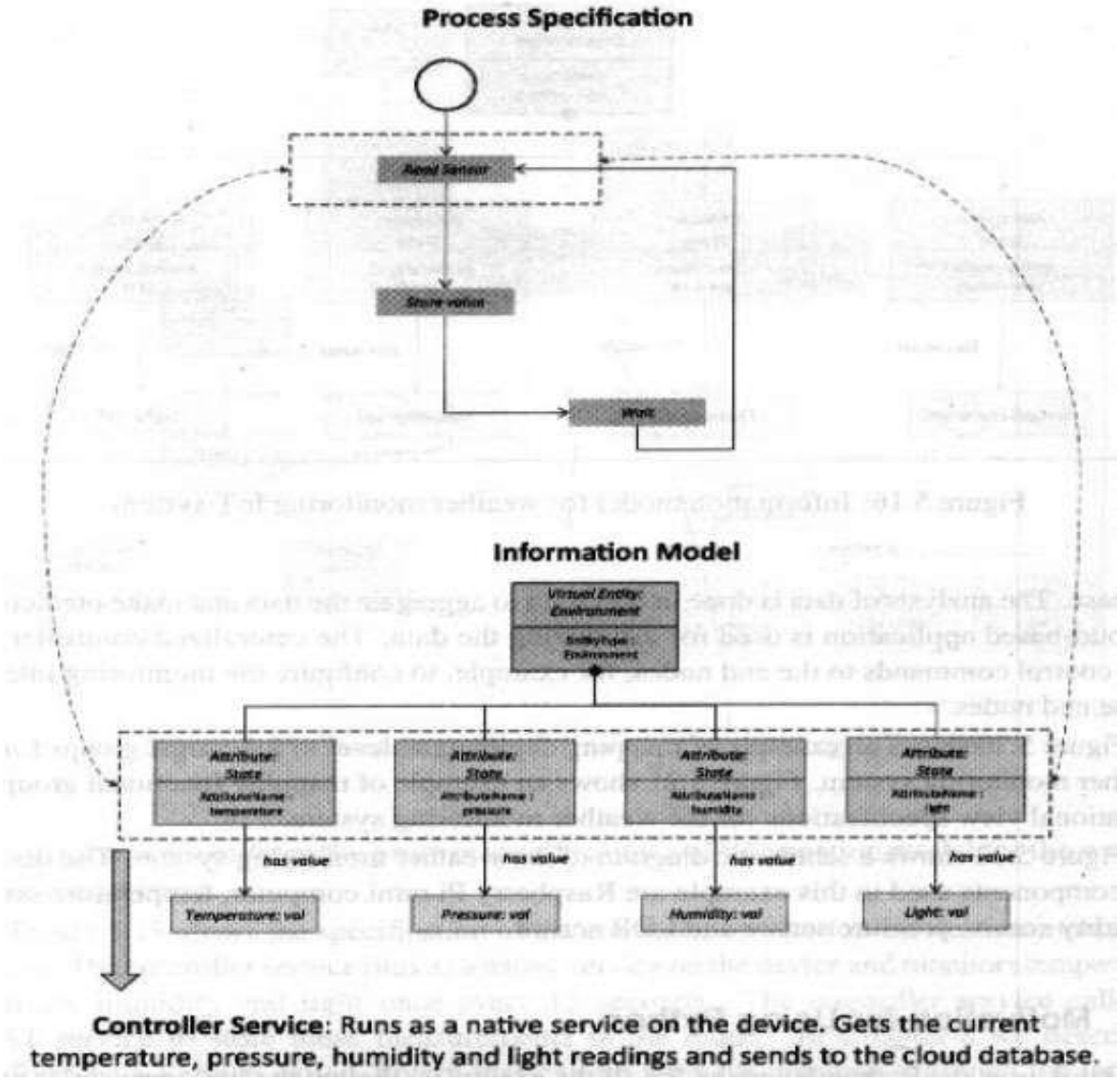
Resources are software components which can be either on-device or network-resources. Services include the controller service that monitors the temperature, pressure, humidity and light and sends the readings to the cloud.



The following figure shows the information model for the weather monitoring system. In this example, there is one virtual entity for the environment being sensed. The virtual entity has attributes – temperature, pressure, humidity and light.

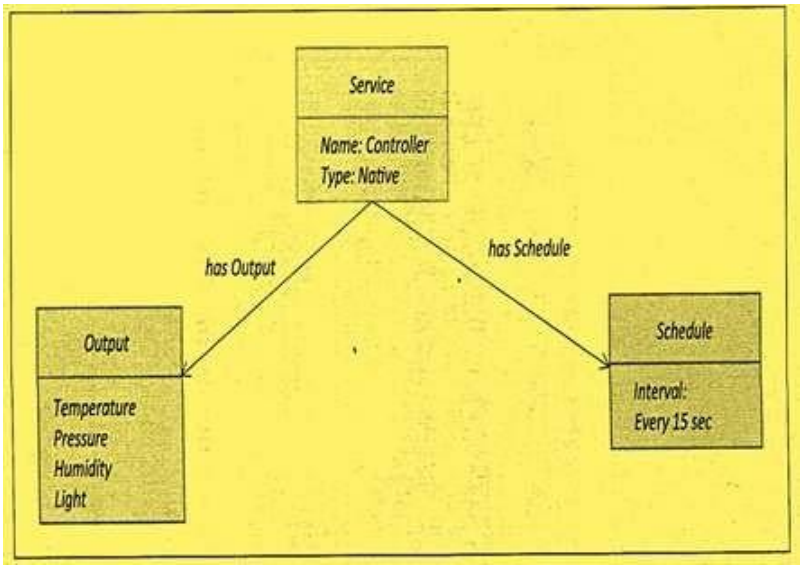


The following figure shows an example of deriving the services from the process specification and information model for the whether monitoring system.

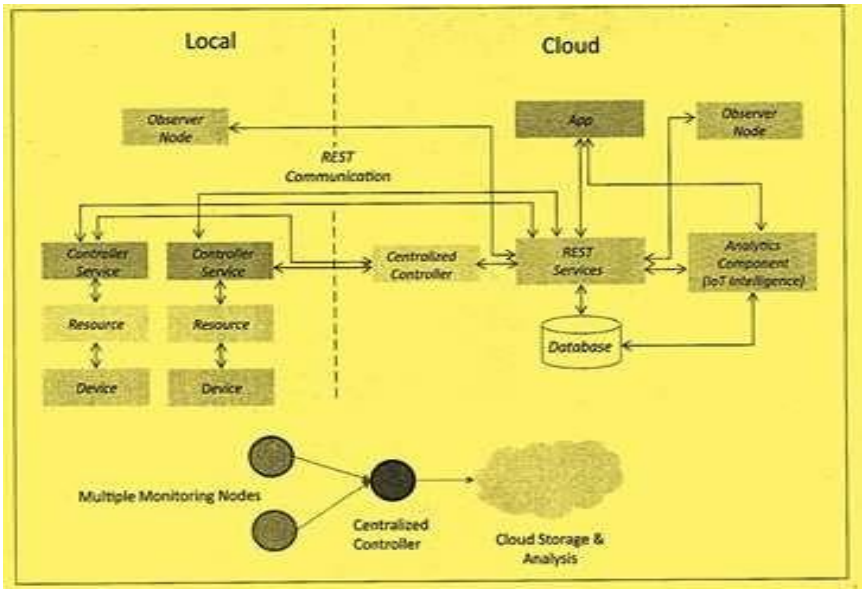


Controller Service: Runs as a native service on the device. Gets the current temperature, pressure, humidity and light readings and sends to the cloud database.

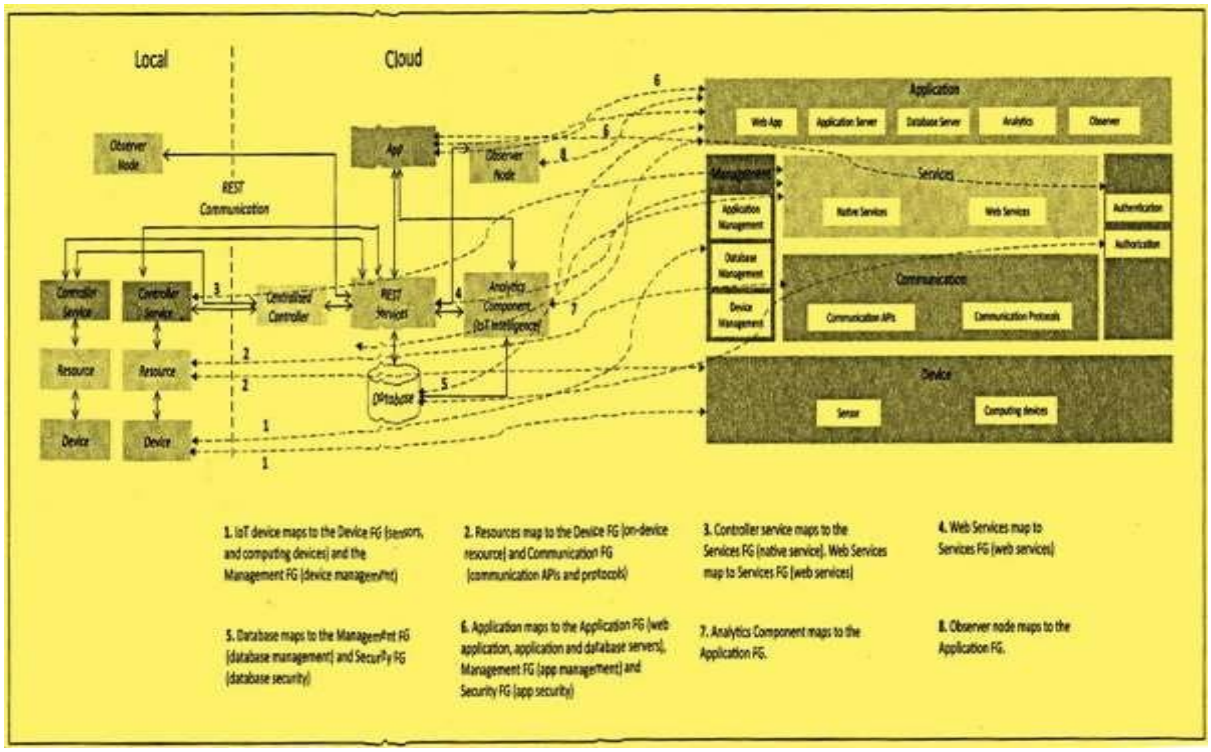
The following figure shows the specification of the controller service for the whether monitoring system. The controller service runs as a native service on the device and monitors temperature, pressure, humidity and light once every 15 seconds. The controller service calls the REST service to store these measurements in the cloud.



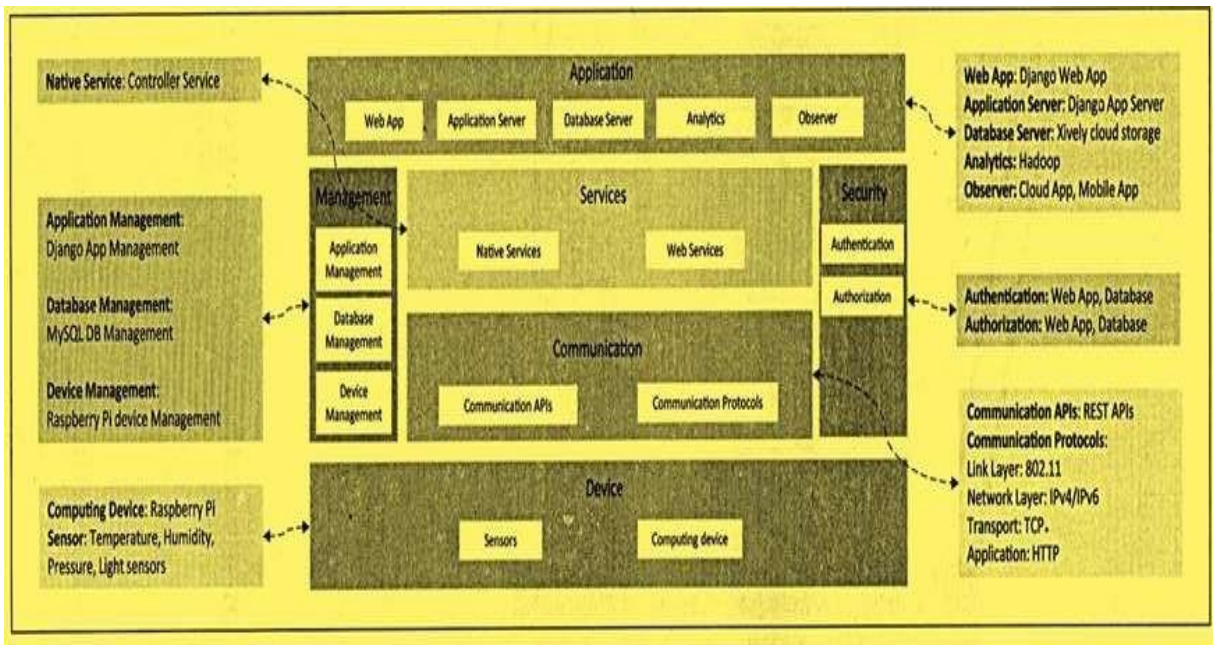
The following figure shows the deployment design for the system. The system consists of multiple nodes placed in different locations for monitoring temperature, humidity and pressure in an area. The end nodes are equipped with various sensors (such as temperature, pressure, humidity and light). The end nodes send the data to the cloud and data is stored in a cloud database. The analysis of data is done in the cloud to aggregate the data and make predictions. A cloud-based application is used for visualizing the data. The centralized controller sends control commands to the end nodes. For example, to configure the monitoring interval on the end nodes.



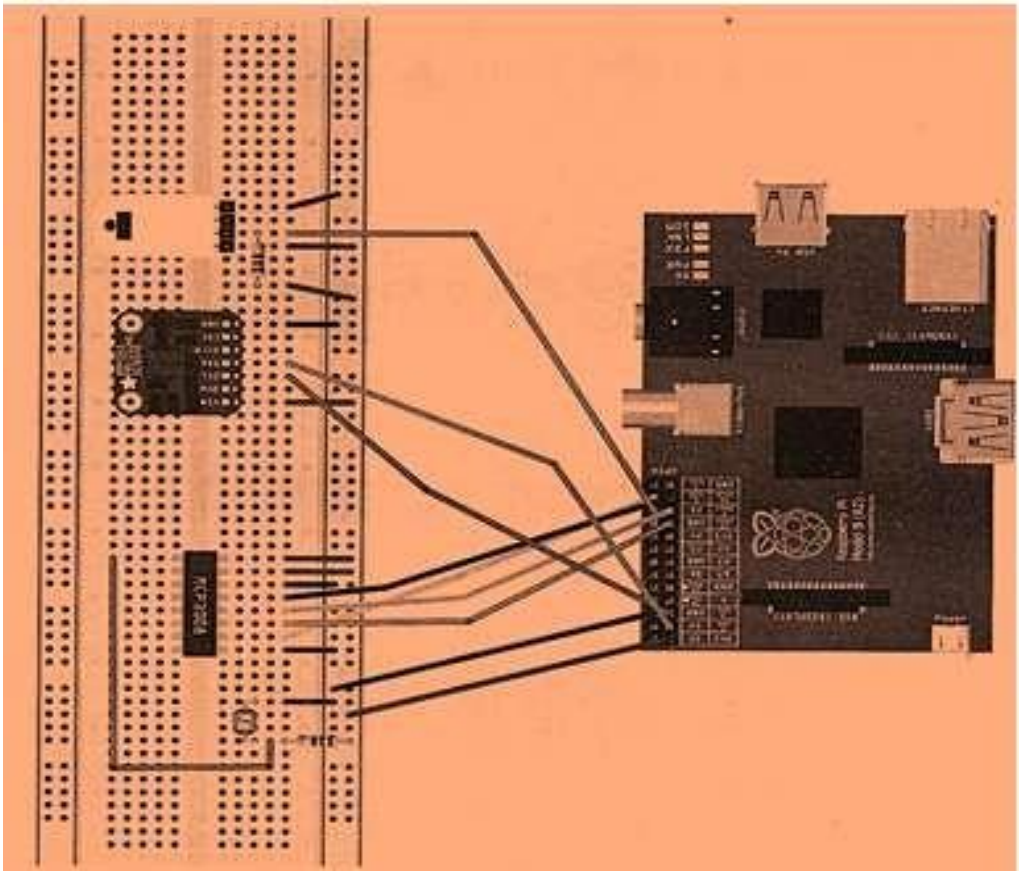
The following figure shows and example of mapping deployment level to functional groups for the whether monitoring system.



The following figure shows an example of mapping functional groups to operational view specifications for the weather monitoring system.



The following figure shows a schematic diagram of the whether monitoring system. The devices and components used in this example are Rasberry Pi minicomputer, temperature, sensor, humidity sensor, pressure sensor and LDR sensor.



Motivation for Using Python:

Python is a minimalistic language with English-like keywords and fewer syntactical constructions as compared to other languages. This makes Python easier to learn and understand. Moreover, Python code compact as compared to other languages. Python is an interpreted language and doesn't require an explicit compilation step. The Python interpreter converts the Python code to the intermediate byte code, specific to the system. Python is supported on wide range of platforms; hence Python code is easily portable.

The wide library support available for Python makes it an excellent choice for IoT systems. Python can be used for end-to-end development of IoT systems from IoT device code (e.g., code for capturing sensor data), native services (e.g., controller service implemented in Python), web applications (e.g., Python web applications developed with Python web frameworks such as Django) and analytics components (e.g., machine learning components developed using Python libraries such as scikit-learn).

IoT Systems – Logical Design using Python:

Introduction to Python:

"Python is a general-purpose high level programming language and suitable for providing a solid foundation to the reader in the area of cloud computing."

The main characteristics of Python are:

1) **Multi-paradigm programming language** - Python supports more than one programming paradigms including object-oriented programming and structured programming
2) **Interpreted Language** - Python is an interpreted language and does not require an explicit compilation step. The Python interpreter executes the program source code directly, statement by statement, as a processor or scripting engine does.
3) **Interactive Language** - Python provides an interactive mode in which the user can submit commands at the Python prompt and interact with the interpreter directly.

The Benefits of Python are:

1) **Easy-to-learn, read and maintain** - Python is a minimalistic language with relatively few keywords, uses English keywords and has fewer syntactical constructions as compared to other languages. Reading Python programs feels like English with pseudo-code like constructs. Python is easy to learn yet an extremely powerful language for a wide range of applications.
2) **Object and Procedure Oriented** - Python supports both procedure-oriented programming and object-oriented programming. Procedure oriented paradigm allows programs to be written around procedures or functions that allow reuse of code. Procedure oriented paradigm allows programs to be written around objects that include both data and functionality.
3) **Extendable** - Python is an extendable language and allows integration of low-level modules written in languages such as C/C++. This is useful when you want to speed up a critical portion of a program.
4) **Scalable** - Due to the minimalistic nature of Python, it provides a manageable structure for large programs.
5) **Portable** - Since Python is an interpreted language, programmers do not have to worry about compilation, linking and loading of programs. Python programs can be directly executed from source.
6) **Broad Library Support** - Python has a broad library support and works on various platforms such as Windows, Linux, Mac, etc.

Installing Python (Python Setup):

Python is a highly portable language that works on various platforms such as Windows, Linux, Mac, etc.

Windows:

1) Python binaries for Windows can be downloaded from http://www.python.org/getit.
2) For the examples and exercise in this book, you would require Python 2.7 which can be directly downloaded from:

   http://www.python.org/ftp/python/2.7.5/python-2.7.5.msi

3) Once the python binary is installed you can run the python shell at the command prompt using

   > python

Linux:

```
#Install Dependencies
sudo apt-get install build-essential
sudo apt-get install libreadline-gplv2-dev libncursesw5-dev libssl-dev libsqlite3-dev tk-dev libgdbm-dev  libc6-dev libbz2-dev

#Download Python
wget http://python.org/ftp/python/2.7.5/Python-2.7.5.tgz
tar -xvf Python-2.7.5.tgz
cd Python-2.7.5

#Install Python
./configure
make
sudo make install
```

Numbers:

Number data type is used to store numeric values. Numbers are immutable data types, therefore changing the value of a number data type results in a newly allocated object.

```
#Integer
>>>a=5
>>>type(a)
<type 'int'>

#Floating Point
>>>b=2.5
>>>type(b)
<type 'float'>

#Long
>>>x=9898878787676L
>>>type(x)
<type 'long'>

#Complex
>>>y=2+5j
>>>y
(2+5j)
>>>type(y)
<type 'complex'>
>>>y.real
2
>>>y.imag
5
```

```
#Addition
>>>c=a+b
>>>c
7.5
>>>type(c)
<type 'float'>

#Subtraction
>>>d=a-b
>>>d
2.5
>>>type(d)
<type 'float'>

#Multiplication
>>>e=a*b
>>>e
12.5
>>>type(e)
<type 'float'>
```

```
#Division
>>>f=b/a
>>>f
0.5
>>>type(f)
<type float'>

#Power
>>>g=a**2
>>>g
25
```

Strings:

A string is simply a list of characters in order. There are no limits to the number of characters you can have in a string.

```
#Create string
>>>s="Hello World!"
>>>type(s)
<type 'str'>

#String concatenation
>>>t="This is sample program."
>>>r = s+t
>>>r
'Hello World!This is sample program.'

#Get length of string
>>>len(s)
12

#Convert string to integer
>>>x="100"
>>>type(s)
<type 'str'>
>>>y=int(x)
>>>y
100
```

```
#Print string
>>>print s
Hello World!

#Formatting output
>>>print "The string (The string (Hello World!)
has 12 characters

#Convert to upper/lower case
>>>s.upper()
'HELLO WORLD!'
>>>s.lower()
'hello world!'

#Accessing sub-strings
>>>s[0]
'H'
>>>s[6:]
'World!'
>>>s[6:-1]
'World'
```

```
#strip: Returns a copy of the string with the
#leading and trailing characters removed.

>>>s.strip("!")
'Hello World'
```

Lists:

List a compound data type used to group together other values. List items need not all have the same type. A list contains items separated by commas and enclosed within square brackets.

```
#Create List
>>>fruits=['apple','orange','banana','mango']
>>>type(fruits)
<type 'list'>

#Get Length of List
>>>len(fruits)
4

#Access List Elements
>>>fruits[1]
'orange'
>>>fruits[1:3]
['orange', 'banana']
>>>fruits[1:]
['orange', 'banana', 'mango']

#Appending an item to a list
>>>fruits.append('pear')
>>>fruits
['apple', 'orange', 'banana', 'mango', 'pear']
```

```
#Removing an item from a list
>>>fruits.remove('mango')
>>>fruits
['apple', 'orange', 'banana', 'pear']

#Inserting an item to a list
>>>fruits.insert(1,'mango')
>>>fruits
['apple', 'mango', 'orange', 'banana', 'pear']

#Combining lists
>>>vegetables=['potato','carrot','onion','beans','r
adish']
>>>vegetables
['potato', 'carrot', 'onion', 'beans', 'radish']

>>>eatables=fruits+vegetables
>>>eatables
['appl
e',
'mang
o',
'orang
e',
'banan
a',
'pear', 'potato', 'carrot', 'onion', 'beans', 'radish']
```

```
#Mixed data types in a list
>>>mixed=['data',5,100.1,8287398L]
>>>type(mixed)
<type 'list'>
>>>type(mixed[0])
<type 'str'>
>>>type(mixed[1])
<type 'int'>
>>>type(mixed[2])
<type 'float'>
>>>type(mixed[3])
<type 'long'>

#Change individual elements of a list
>>>mixed[0]=mixed[0]+" items"
>>>mixed[1]=mixed[1]+1
>>>mixed[2]=mixed[2]+0.05
>>>mixed
['data items', 6, 100.14999999999999, 8287398L]

#Lists can be nested
>>>nested=[fruits,vegetables]
>>>nested
[['apple', 'mango', 'orange', 'banana', 'pear'],
['potato', 'carrot', 'onion', 'beans', 'radish']]
```

Tuples:

A tuple is a sequence data type that is similar to the list. A tuple consists of a number of values separated by commas and enclosed within parentheses. Unlike lists, the elements of tuples cannot be changed, so tuples can be thought of as read-only lists.

```
#Create a Tuple
>>>fruits=("apple","mango","banana","pineapple")
>>>fruits
('apple', 'mango', 'banana', 'pineapple')

>>>type(fruits)
<type 'tuple'>

#Get length of tuple
>>>len(fruits)
4
```

```
#Get an element from a tuple
>>>fruits[0]
'apple'
>>>fruits[:2]
('apple', 'mango')

#Combining tuples
>>>vegetables=('potato','carrot','onion','radish')
>>>eatables=fruits+vegetables
>>>eatables
('apple', 'mango', 'banana', 'pineapple', 'potato', 'carrot', 'onion', 'radish')
```

Dictionaries:

Dictionary is a mapping data type or a kind of hash table that maps keys to values. Keys in a dictionary can be of any data type, though numbers and strings are commonly used for keys. Values in a dictionary can be any data type or object.

```
#Create a dictionary
>>>student={'name':'Mary','id':'8776','major':'CS'}
>>>student
{'major': 'CS', 'name': 'Mary', 'id': '8776'}
>>>type(student)
<type 'dict'>

#Get length of a dictionary
>>>len(student)
3

#Get the value of a key in dictionary
>>>student['name']
'Mary'

#Get all items in a dictionary
>>>student.items()
[('gender', 'female'), ('major', 'CS'), ('name', 'Mary'),
('id', '8776')]
```

```
#Get all keys in a dictionary
>>>student.keys()
['gender', 'major', 'name', 'id']

#Get all values in a dictionary
>>>student.values()
['female', 'CS', 'Mary', '8776']

#Add new key-value pair
>>>student['gender']='female'
>>>student
{'gende
r': 'female', 'major': 'CS', 'name': 'Mary', 'id': '8776'}

#A value in a dictionary can be another dictionary
>>>student1={'name':'David','id':'9876','major':'ECE'}
>>>students={'1': student,'2':student1}
>>>students
{'1':
{'gende
r': 'female', 'major': 'CS', 'name': 'Mary', 'id': '8776'}, '2':
{'
major': 'ECE', 'name': 'David', 'id': '9876'}}
```

```
#Check if dictionary has a key
>>>student.has_key('name')
True
>>>student.has_key('grade')
False
```

Type Conversions:

To convert from one data type to another is known as Type Conversion.

```
#Convert to string
>>>a=10000
>>>str(a)
'10000'

#Convert to int
>>>b="2013"
>>>int(b)
2013

#Convert to float
>>>float(b)
2013.0
```

```
#Convert to long
>>>long(b)
2013L

#Convert to list
>>>s="aeiou"
>>>list(s)
['a', 'e', 'i', 'o', 'u']

#Convert to set
>>>x=['mango','apple','banana','mango','banana']
>>>set(x)
set(['mango', 'apple', 'banana'])
```

Control Flow – if statement:

The if statement in Python is similar to the if statement in other languages.

```
>>>a = 25**5
>>>if a>10000:
        print "More"
    else:
        print "Less"

More
```

```
>>>s="Hello World"
>>>if "World" in s:
        s=s+"!"
print s

Hello World!
```

```
>>>if a>10000:
    if a<1000000:
        print "Between 10k and 100k"
    else:
        print "More than 100k"
    elif a==10000:
        print "Equal to 10k"
    else:
        print "Less than 10k"

More than 100k
```

```
>>>student={'name':'Mary','id':'8776'}
>>>if not student.has_key('major'):
        student['major']='CS'

>>>student
{'major': 'CS', 'name': 'Mary', 'id': '8776'}
```

Control Flow – for statement:

1) The for statement in Python iterates over items of any sequence (list, string, etc.) in the order in which they appear in the sequence.
2) This behavior is different from the for statement in other languages such as C in which an initialization, incrementing and stopping criteria are provided.

```
#Looping over characters in a string

helloString = "Hello World"

for c in helloString:
        print c
```

```
#Looping over items in a list

fruits=['apple','orange','banana','mango']

i=0
for item in fruits:
        print "Fruit-%d: %s" % (i,item)
        i=i+1
```

```
#Looping over keys in a dictionary

student
=
'nam
e':
'Mar
y', 'id': '8776','gender': 'female', 'major': 'CS'

for key in student:
        print "%s: %s" % (key,student[key]
```

Control Flow – while statement:

The while statement in Python executes the statements within the while loop as long as the while condition is true.

```
#Prints even numbers upto 100

>>> i = 0

>>> while i<=100:
if i%2 == 0:
    print i
i = i+1
```

Control Flow – range statement:

The range statement in Python generates a list of numbers in arithmetic progression.

```
#Generate a list of numbers from 0 – 9

>>>range (10)
[0, 1, 2, 3, 4, 5, 6, 7, 8, 9]
```

```
#Generate a list of numbers from 10 - 100 with increments of 10

>>>range(10,110,10)
[10, 20, 30, 40, 50, 60, 70, 80, 90,100]
```

Control Flow – break/continue statements:

The break and continue statements in Python are similar to the statements in C.

Break: The "break" statement breaks out of the for/while loop.

```
#Break statement example
>>>y=1
>>>for x in range(4,256,4):
        y = y * x
        if y > 512:
                break
        print y
4
32
384
```

Continue: The ”continue” statement continues with the next iteration.

```
#Continue statement example
>>>fruits=['apple','orange','banana','mango']
>>>for item in fruits:
        if item == "banana":
                continue
        else:
                print item

apple
orange
mango
```

Control Flow – pass statement:

1) The pass statement in Python is a null operation.
2) The pass statement is used when a statement is required syntactically but you do not want any command or code to execute.

```
>fruits=['apple','orange','banana','mango']
>for item in fruits:
        if item == "banana":
                pass
        else:
                print item

apple
orange
mango
```

Functions:

1) A function is a block of code that takes information in (in the form of parameters), does some computation, and returns a new piece of information based on the parameter information.
2) A function in Python is a block of code that begins with the keyword def followed by the function name and parentheses. The function parameters are enclosed within the parenthesis.
3) The code block within a function begins after a colon that comes after the parenthesis enclosing the parameters.
4) The first statement of the function body can optionally be a documentation string or docstring.

```
students = { '1': {'name': 'Bob', 'grade': 2.5},
             '2': {'name': 'Mary', 'grade': 3.5},
             '3': {'name': 'David', 'grade': 4.2},
             '4': {'name': 'John', 'grade': 4.1},
             '5': {'name': 'Alex', 'grade': 3.8}}
```

```
def averageGrade(students):
        "This function computes the average grade"
        sum = 0.0
        for key in students:
                sum = sum + students[key]['grade']
                average = sum/len(students)
        return average


avg = averageGrade(students)
print "The average garde is: %0.2f" % (avg)
```

Functions - Default Arguments:

1) Functions can have default values of the parameters.
2) If a function with default values is called with fewer parameters or without any parameter, the default values of the parameters are used.

```
>>>def displayFruits(fruits=['apple','orange']):
        print "There are %d fruits in the list" % (len(fruits))
        for item in fruits:
                print item

#Using default arguments
>>>displayFruits()
apple
orange


>>>fruits = ['banana', 'pear', 'mango']
>>>displayFruits(fruits)
banana
pear
mango
```

Functions - Passing by Reference:

1) All parameters in the Python functions are passed by reference.
2) If a parameter is changed within a function the change also reflected back in the calling function.

```
>>>def displayFruits(fruits):
        print "There are %d fruits in the list" % (len(fruits))
        for item in fruits:
                print item
        print "Adding one more fruit"
        fruits.append('mango')

>>>fruits = ['banana', 'pear', 'apple']
>>>displayFruits(fruits)
There are 3 fruits in the list
banana
pear
apple

#Adding one more fruit
>>>print "There are %d fruits in the list" % (len(fruits))
There are 4 fruits in the list
```

Functions - Keyword Arguments:

Functions can also be called using keyword arguments that identifies the arguments by the parameter name when the function is called.

```
>>>def
printStudentRecords(name,age=20,major='CS'):
        print "Name: " + name
        print "Age: " + str(age)
        print "Major: " + major

#This will give error as name is required argument
>>>printStudentRecords()
Traceback (most recent call last):
File "<stdin>", line 1, in <module>
TypeError: printStudentRecords() takes at least 1
argument (0 given)
```

```
#Correct use
>>>printStudentRecords(name='Alex')
Name: Alex
Age: 20
Major: CS

>>>printStudentRecords(name='Bob',age=22,major='EC
E')
Name: Bob
Age: 22
Major: ECE

>>>printStudentRecords(name='Alan',major='ECE')
Name: Alan
Age: 20
Major: ECE
```

```
#name is a formal argument.
#**kwargs is a keyword argument that receives all
arguments except the formal argument as a
dictionary.

>>>def student(name, **kwargs):
        print "Student Name: " + name
        for key in kwargs:
            print key + ': ' + kwargs[key]

>>>student(name='Bob', age='20', major = 'CS')
Student Name: Bob
age: 20
major: CS
```

Functions - Variable Length Arguments:

Python functions can have variable length arguments. The variable length arguments are passed to as a tuple to the function with an argument prefixed with asterix (*).

```
>>>def student(name, *varargs):
        print "Student Name: " + name
            for item in varargs:
        print item

>>>student('Nav')
Student Name: Nav

>>>student('Amy', 'Age: 24')
Student Name: Amy
Age: 24

>>>student('Bob', 'Age: 20', 'Major: CS')
Student Name: Bob
Age: 20
Major: CS
```

Modules:

1) Python allows organizing the program code into different modules which improves the code readability and management.
2) A module is a Python file that defines some functionality in the form of functions or classes.
3) Modules can be imported using the import keyword.
4) Modules to be imported must be present in the search path.

```
#student module - saved as student.py
def averageGrade(students):
        sum = 0.0
        for key in students:
           sum = sum + students[key]['grade']
           average = sum/len(students)
        return average

def printRecords(students):
        print "There are %d students" %(len(students))
        i=1
        for key in students:
           print "Student-%d: " % (i)
           print "Name: " + students[key]['name']
           print "Grade: " + str(students[key]['grade'])
           i = i+1
```

```
# Importing a specific function from a module
>>>from student import averageGrade

# Listing all names defines in a module
>>>dir(student)
```

```
#Using student module
>>>import student
>>>students =  '1': 'name': 'Bob', 'grade': 2.5,
'2': 'name': 'Mary', 'grade': 3.5,
'3': 'name': 'David', 'grade': 4.2,
'4': 'name': 'John', 'grade': 4.1,
'5': 'name': 'Alex', 'grade': 3.8

>>>student.printRecords(students)
There are 5 students
Student-1:
Name: Bob
Grade: 2.5
Student-2:
Name: David
Grade: 4.2
Student-3:
Name: Mary
Grade: 3.5
Student-4:
Name: Alex
Grade: 3.8
Student-5:
Name: John
Grade: 4.1

>>>avg = student. averageGrade(students)
>>>print "The average garde is: %0.2f" % (avg)
3.62
```

Packages:

1) Python package is hierarchical file structure that consists of modules and sub-packages.
2) Packages allow better organization of modules related to a single application environment.

```
# skimage package listing

skimage/                    Top level package
        __init__.py         Treat directory as a package

        color/ color        color subpackage
                __init__.py
                colorconv.py
                colorlabel.py
                rgb_colors.py

        draw/ draw          draw subpackage
                __init__.py
                draw.py
                setup.py

        exposure/           exposure subpackage
                __init__.py
                _adapthist.py
                exposure.py

        feature/            feature subpackage
                __init__.py
                _brief.py
                _daisy.py
...
```

File Handling:

1) Python allows reading and writing to files using the file object.
2) The open(filename, mode) function is used to get a file object.
3) The mode can be read (r), write (w), append (a), read and write (r+ or w+), read-binary (rb), write-binary (wb), etc.
4) After the file contents have been read the close function is called which closes the file object.

```
# Example of reading an entire file

>>>fp = open('file.txt','r')
>>>content = fp.read()
>>>print content
This is a test file.
>>>fp.close()
```

```
# Example of reading line by line

>>>fp = open('file1.txt','r')
>>>print "Line-1: " + fp.readline()
Line-1: Python supports more than one programming paradigms.
>>>print "Line-2: " + fp.readline()
Line-2: Python is an interpreted language.
>>>fp.close()
```

```
# Example of reading lines in a loop

>>>fp = open('file1.txt','r')
>>>lines = fp.readlines()
>>>for line in lines:
        print line

Python supports more than one programming paradigms.
Python is an interpreted language.
```

```
# Example of reading a certain number of bytes

>>>fp = open('file.txt','r')
>>>fp.read(10)
'Python sup'
>>>fp.close()
```

```
# Example of seeking to a certain position

>>>fp = open('file.txt','r')
>>>fp.seek(10,0)
>>>content = fp.read(10)
>>>print content
ports more
>>>fp.close()
```

```
# Example of getting the current position of read

>>>fp = open('file.txt','r')
>>>fp.read(10)
'Python sup'
>>>currentpos = fp.tell
>>>print currentpos
<built-in method tell of file object at 0x0000000002391390>
>>>fp.close()
```

```
# Example of writing to a file

>>>fo = open('file1.txt','w')
>>>content='This is an example of writing to a file in
Python.'
>>>fo.write(content)
>>>fo.close()
```

Date/Time Operations:

1) Python provides several functions for date and time access and conversions.
2) The datetime module allows manipulating date and time in several ways.
3) The time module in Python provides various time-related functions.

```
# Examples of manipulating with date

>>>from datetime import date

>>>now = date.today()
>>>print "Date: " + now.strftime("%m-%d-%y")
Date: 07-24-13

>>>print "Day of Week: " + now.strftime("%A")
Day of Week: Wednesday

>>>print "Month: " + now.strftime("%B")
Month: July

>>>then = date(2013, 6, 7)
>>>timediff = now - then
>>>timediff.days
47
```

```
# Examples of manipulating with time

>>>import time
>>>nowtime = time.time()
>>>time.localtime(nowtime)
time.struct_time(tm_year=2013, tm_mon=7, tm_mday=24, tm_ec=51, tm_wday=2, tm_yday=205,
tm_isdst=0)

>>>time.asctime(time.localtime(nowtime))
'Wed Jul 24 16:14:51 2013'

>>>time.strftime("The date is %d-%m-%y. Today is a %A. It is %H hours, %M minutes and %S seconds now.")
'The date is 24-07-13. Today is a Wednesday. It is 16 hours, 15 minutes and 14 seconds now.'
```

Classes:

Python is an Object-Oriented Programming (OOP) language. Python provides all the standard features of Object Oriented Programming such as classes, class variables, class methods, inheritance, function overloading, and operator overloading.

1) **Class -** A class is simply a representation of a type of object and user-defined prototype for an object that is composed of three things: a name, attributes, and operations/methods.
2) **Instance/Object -** Object is an instance of the data structure defined by a class.
3) **Inheritance -** Inheritance is the process of forming a new class from an existing class or base class.

4) **Function overloading -** Function overloading is a form of polymorphism that allows a function to have different meanings, depending on its context.
5) **Operator overloading -** Operator overloading is a form of polymorphism that allows assignment of more than one function to a particular operator.
6) **Function overriding -** Function overriding allows a child class to provide a specific implementation of a function that is already provided by the base class.

Ex:

```
# Examples of a class
class Student:
        studentCount = 0

        def __init__(self, name, id):
                print "Constructor called"
                self.name = name
                self.id = id
                Student.studentCount = Student.studentCount + 1
                self.grades={}

        def __del__(self):
                print "Destructor called"

        def getStudentCount(self):
                return Student.studentCount

        def addGrade(self,key,value):
                self.grades[key]=value
                def getGrade(self,key):
                return self.grades[key]

        def printGrades(self):
                for key in self.grades:
                        print key + ": " + self.grades[key]
```

```
>>>s = Student('Steve','98928')
Constructor called

>>>s.addGrade('Math','90')
>>>s.addGrade('Physics','85')
>>>s.printGrades()
Physics: 85
Math: 90

>>>mathgrade = s.getGrade('Math')
>>>print mathgrade
90

>>>count = s.getStudentCount()
>>>print count
1

>>>del s
Destructor called
```

a) The variable studentCount is a class variable that is shared by all instances of the class Student and is accessed by Student.studentCount.
b) The variables name, id and grades are instance variables which are specific to each instance of the class.
c) There is a special method by the name __init__() which is the class constructor.
d) The class constructor initializes a new instance when it is created. The function __del__() is the class destructor.

Class Inheritance:

a) In this example Shape is the base class and Circle is the derived class. The class Circle inherits the attributes of the Shape class.

b) The child class Circle overrides the methods and attributes of the base class (eg. draw() function defined in the base class Shape is overridden in child class Circle).

```
# Examples of class inheritance
class Shape:
        def __init__(self):
                print "Base class constructor"
                self.color = 'Green'
                self.lineWeight = 10.0

        def draw(self):
                print "Draw - to be implemented"
                def setColor(self, c):
                self.color = c
                def getColor(self):
                return self.color

        def setLineWeight(self,lwt):
                self.lineWeight = lwt

        def getLineWeight(self):
                return self.lineWeight
```

```
class Circle(Shape):
        def __init__(self, c,r):
                print "Child class constructor"
                self.center = c
                self.radius = r
                self.color = 'Green'
                self.lineWeight = 10.0
                self.__label = 'Hidden circle label'

        def setCenter(self,c):
                self.center = c
                def getCenter(self):
                return self.center

        def setRadius(self,r):
                self.radius = r

        def getRadius(self):
                return self.radius

        def draw(self):
                print "Draw Circle (overridden function)"
```

```
class Point:
        def __init__(self, x, y):
                self.xCoordinate = x
                self.yCoordinate = y

        def setXCoordinate(self,x):
                self.xCoordinate = x

        def getXCoordinate(self):
                return self.xCoordinate

        def setYCoordinate(self,y):
                self.yCoordinate = y

        def getYCoordinate(self):
                return self.yCoordinate
```

```
>>>p = Point(2,4)
>>>circ = Circle(p,7)
Child class constructor
>>>circ.getColor()
'Green'
>>>circ.setColor('Red')
>>>circ.getColor()
'Red'
>>>circ.getLineWeight()
10.0
>>>circ.getCenter().getXCoordinate()
2
>>>circ.getCenter().getYCoordinate()
4
>>>circ.draw()
Draw Circle (overridden function)
>>>circ.radius
7
```

Python Packages of Interest for IoT:

1. Java Script Object Notation (JSON): This is an easy to read and write data-interchange format. JSON is used as an alternative to XML and is easy for machines to parse and generate. JSON is built on two structures – a collection of name-value pairs (e.g. Python dictionary) and ordered list of values (e.g. Python list).

JSON format is often used for serializing and transmitting structured data over a network connection. For example, transmitting data between a server and web application. The following code is an example of a Twitter tweet object encoded as JSON.

JSON Example - A Twitter tweet object

```
{
   "created_at":
   "Sat Jun 01 11:39:43 +0000 2013",
   "id":340794787059875841,
   "text":"What a bright and sunny day today!",
   "truncated":false,
   "in_reply_to_status_id":null,
   "user":{
      "id":383825039,
      "name":"Harry",
      "followers_count":316,
      "friends_count":298,
      "listed_count":0,
      "created_at":"Sun Oct 02 15:51:16 +0000 2011",
      "favorites_count":251,
      "statuses_count":1707,
      :
      "notifications":null
      },
   "geo":{
      "type":"Point",
      "coordinates":[26.92782727,75.78908449]
      },
   "coordinates":{
      "type":"Point",
      "coordinates":[75.78908449,26.92782727]
      },
   "place":null,
   "contributors":null,
   "retweet_count":0,
   "favorite_count":0,
   "entities":{
      "hashtags":[],
      "symbols":[],
      "urls":[],
      "user_mentions":[]
      },
   "favorited":false,
   "retweeted":false,
   "filter_level":"medium",
   "lang":"nl"
}
```

Exchange of information encoded as JSON involves encoding and decoding. The Python JSON package [109] provides functions for encoding and decoding JSON. The following code shows an example of encoding and decoding.

Encoding & Decoding JSON in Python

```
>>>import json

>>>message = {
    "created":  "Wed Jun 31 2013",
    "id":"001",
    "text":"This is a test message.",
}

>>>json.dumps(message)
'{"text":  "This is a test message.", "id":  "001",
"created":  "Wed Jun 31 2013"}'


>>>decodedMsg = json.loads( '{"text":  "This is a
test message.", "id":  "001", "created":  "Wed Jun 31 2013"}')

>>>decodedMsg['created']
u'Wed Jun 31 2013'
>>>decodedMsg['text']
u'This is a test message.'
```

2. Extensible Markup Language (XML): This is a data format for structured document interchange. The following code shows an example of an XML file.

XML example

```
<?xml version="1.0"?>
<catalog>
< plant id='1' >
<common>Bloodroot</common>
<botanical>Sanguinaria canadensis</botanical>
<zone>4</zone>
<light>Mostly Shady</light>
<price> 2.44 </price>
<availability>031599</availability>
</plant>
<plant id='2' >
<common>Columbine</common>
<botanical>Aquilegia canadensis</botanical>
<zone>3</zone>
<light>Mostly Shady</light>
<price> 9.37</price >
<availability>030699</availability>
</plant>
<plant id='3' >
<common>Marsh Marigold</common>
<botanical>Caltha palustris</botanical>
<zone>4</zone>
<light>Mostly Sunny</light>
<price> 6.81</price>
<availability>051799</availability>
</plant>
</catalog>
```

The Python "minidom" library provides a minimal implementation of the Document Object Model (DOM) interface and has an API similar to that in other languages. It is shown in the following code.

Parsing an XML file in Python

```
from xml.dom.minidom import parse
dom = parse("test.xml")
for node in dom.getElementsByTagName('plant'):
   id=node.getAttribute('id')
   print "Plant ID:", id
   common=node.getElementsByTagName('common')[0]
      .childNodes[0].nodeValue
   print "Common:", common
   botanical=node.getElementsByTagName('botanical')[0]
      .childNodes[0].nodeValue
   print "Botanical:", botanical
   zone=node.getElementsByTagName('zone')[0]
      .childNodes[0].nodeValue
   print "Zone:", zone
```

The following code explains Python program for creating an XML file.

Creating an XML file with Python

```
#Python example to create the following XML:
#' <?xml version="1.0" ?> <Class> <Student>
#<Name>Alex</Name> <Major>ECE</Major> </Student > </Class>

from xml.dom.minidom import Document
doc = Document()

# create base element
base = doc.createElement('Class')
doc.appendChild(base)

# create an entry element
entry = doc.createElement('Student')
base.appendChild(entry)

# create an element and append to entry element
name = doc.createElement('Name')
nameContent = doc.createTextNode('Alex')
name.appendChild(nameContent)
entry.appendChild(name)

# create an element and append to entry element
major = doc.createElement('Major')
majorContent = doc.createTextNode('ECE')
major.appendChild(majorContent)
entry.appendChild(major)

fp = open('foo.xml','w')
doc.writexml()
fp.close()
```

3. HTTPLib & URLLib: These two are Python libraries used in network/internet programming [111, 112]. HTTPLib2 is an HTTP client library and URLLib2 is a library for fetching URLs. The following code in an example of an HTTP GET request using the HTTPLib. The variable ***resp*** contains the response headers and ***content*** contains the content retrieved from the URL.

HTTP GET request example using HTTPLib

```
>>> import httplib2
>>> h = httplib2.Http()
>>> resp, content = h.request("http://example.com", "GET")
>>> resp
{'status': '200', 'content-length': '1270', 'content-location':
'http://example.com', 'x-cache': 'HIT', 'accept-ranges':
'bytes', 'server': 'ECS
(cpm/F858)', 'last-modified': 'Thu,
25 Apr 2013 16:13:23 GMT', 'etag':
'"780602-4f6-4db31b2978ec0"', 'date': 'Wed, 31 Jul 2013 12:36:05 GMT',
'content-type': 'text/html; charset=UTF-8'}

>>> content
'<!doctype html>\n<html>\n<head>\n
<title>Example Domain</title>\n\n
<meta charset="utf-8" />\n
:
```

The following code explains an HTTP request example using URLLib2. A request object is created by calling ***urllib2.Request*** with the URL to fetch as input parameter. Then ***urllib2.urlopen*** is called with the request object which returns the response object for the requested URL. The response object is read by calling ***read*** function.

HTTP request example using URLLib2

```
>>> import urllib2
>>>
>>> req = urllib2.Request('http://example.com')
>>> response = urllib2.urlopen(req)
>>> response_page = response.read()
>>> response_page
'<!doctype html>\n<html>\n<head>\n
<title>Example Domain</title>\n\n
<meta charset="utf-8" />\n
```

The following code is an example of an HTTP POST request. The data in the POST body is encoded using the ***urlencode*** function from ***urllib***.

HTTP POST example using HTTPLib2

```
>>> import httplib2
>>> import urllib
>>> h = httplib2.Http()
>>> data = {'title':  'Cloud computing'}
>>> resp, content =
h.request("http://www.htmlcodetutorial.com/cgi-bin/mycgi.pl", "POST",
urllib.urlencode(data))
>>> resp
{'status':  '200', 'transfer-encoding':  'chunked',
'server':  'Apache/2.0.64 (Unix) mod_ssl/2.0.64 OpenSSL/0.9.7a
mod_auth_passthrough/2.1 mod_bwlimited/1.4 FrontPage/5.0.2.2635
PHP/5.3.10', 'connection':  'close', 'date':  'Wed, 31 Jul 2013
12:41:20 GMT', 'content-type':  'text/html; charset=ISO-8859-1'}

>>> content
'<HTML>\n<HEAD>\n<TITLE>Idocs Guide to
HTML: My CGI</TITLE>\n</HEAD>
:
```

The following code is an example of sending data to a URL using URLLib2 (e.g. an HTML form submission). This example is similar to the HTTP POST example and uses URLLib2 request object instead of HTTPLib2.

Example of sending data to a URL

```
>>> import urllib
>>> import urllib2
>>>
>>> url = 'http://www.htmlcodetutorial.com/cgi-bin/mycgi.pl'
>>> values = {'title' :  'Cloud Computing',
...  'language' :  'Python' }
>>>
>>> data = urllib.urlencode(values)
>>> req = urllib2.Request(url, data)
>>> response = urllib2.urlopen(req)
>>> the_page = response.read()
>>> the_page
'<HTML>\n<HEAD>\n<TITLE>Idocs Guide to
HTML: My CGI</TITLE>\n</HEAD>
:
```

4. SMTPLib: Simple Mail Transfer Protocol (SMTP) is a protocol which handles sending email and routing email between mail servers. The Python ***smtplib*** module provides an SMTP client session object that can be used to send email [113]. The following code shows a Python example of sending email from Gmail account. The string ***message*** contains the email message to be sent. To send email from Gmail account, the Gmail SMTP server is specified in the ***server*** string.

To send an email, first a connection is established with the SMTP server by calling ***smtplib.SMTP*** with the SMTP server name and port. The username and password provided are then used to login into the server. The email is then sent by calling ***server.sendmail*** function with the from address, to address list and message as input parameters.

Python example of sending email

```
import smtplib

from_email = '<enter-gmail-address>'
recipients_list = ['<enter-sender-email>']
cc_list = []
subject = 'Hello'
message = 'This is a test message.'
username = '<enter-gmail-username>'
password = '<enter-gmail-password>'
server = 'smtp.gmail.com:587'

def sendemail(from_addr, to_addr_list, cc_addr_list,
subject, message,
login, password,
smtpserver):

header = 'From:  %s\n' % from_addr
header += 'To:  %s\n' % ','.join(to_addr_list)
header += 'Cc:  %s\n' % ','.join(cc_addr_list)
header += 'Subject:  %s\n\n' % subject
message = header + message

server = smtplib.SMTP(smtpserver)
server.starttls()
server.login(login,password)
problems = server.sendmail(from_addr, to_addr_list, message)
server.quit()

#Send email
sendemail(from_email, recipients_list, cc_list, subject,
message, username, password, server)
```

****