Module Title: INTERNET OF THINGS

Module Code: ETT303

By NIYITEGEKA JANVIER

RP/IPRC TUMBA

jniyitegeka@iprctumba.rp.ac.rw

Table of Contents

UNIT 1: INTRODUCTION TO INTERNET OF THINGS (IOT)

1.0 Introduction

The Internet provides the infrastructure designed for global connectivity supporting web browsing, social media, video streaming, collaboration, gaming, online library and online learning amongst others. Normally, when people use the term Internet, they refer to the content accessible online and not the connection of physical things in the real world such as cars, pets, home appliances, trees, etc. The Internet of things aims at connecting everyday objects to the internet through the use of sensors. Kevin Ashton, cofounder and executive director of the Auto-ID Center at MIT, first mentioned the Internet of Things in a presentation he made to Procter & Gamble in 1999. Here's how Ashton explains the potential of the Internet of Things:

- "*Today computers -- and, therefore, the internet -- are almost wholly dependent on human beings for information. Nearly all of the roughly 50 petabytes (a petabyte is 1,024 terabytes) of data available on the internet were first captured and created by human beings by typing, pressing a record button, taking a digital picture or scanning a bar code.*
- *The problem is, people have limited time, attention and accuracy -- all of which means they are not very good at capturing data about things in the real world. If we had computers that knew everything there was to know about things -- using data they gathered without any help from us -- we would be able to track and count everything and greatly reduce waste, loss and cost. We would know when things needed replacing, repairing or recalling and whether they were fresh or past their best."*

Machine to Machine (M2M) communication is the founding concept from which IoT has risen. The Internet of Things (IoT) is a system of interrelated computing devices, mechanical and digital machines, objects, animals or people that are provided with unique identifiers and the ability to transfer data over a network without requiring human-to-human or human-to-computer interaction.

On the personal level, Internet of Things paints a picture of a future world where all the things in our ambient environment are connected to the Internet and seamlessly communicate with each other to operate intelligently. The ultimate (optimum) goal of IoT is to enable objects around us to efficiently sense our surroundings, inexpensively communicate, and ultimately(finally) create a better environment for us.

IoT's promises for business include leveraging automatic sensing and prompt analysis of thousands of service or product-related parameters and then automatically taking action before a service experience or product operation is impacted. It also includes collecting and analyzing massive amounts of structured and unstructured data from various internal and external sources, such as social media, for the purpose of gaining competitive advantage by offering better services and improving business processes.

The term "Internet of Things" was first coined (Invented) by Kevin Ashton in a presentation that he made at Procter & Gamble in 1999. He has mentioned, "The Internet of Things has the potential to change the world, just as the Internet did. Maybe even more so." Afterward, the MIT Auto-ID center presented their IoT vision in 2001. Later, IoT was formally introduced by the International Telecommunication Union (ITU) Internet Report in 2005.

The main power of IoT is the high impact it is already starting to have on business and personal lives. Companies are already employing IoT to create new business models, improve business processes, and reduce costs and risks. Personal lives are improving with advanced health monitoring, enhanced learning, and improved security just to name few examples of possible applications.

IoT has evolved from the convergence of wireless technologies, micro-electromechanical systems (MEMS), micro services and the internet. The convergence has helped to tear down the walls between operational technologies (OT) and information technology (IT), allowing unstructured machine-generated data to be analyzed for insights that will drive improvements.

IoT is built on of four elements:

- **Things –** Physical devices and objects connected to the internet and each other for intelligent decision-making. These objects contain embedded technology that makes

allows them to communicate with IT devices/servers via the network. Example of devices are mobile phone, tablets, Google glass, smart watch, smart TV, smart car, etc.

- **Data** – Data is captured by sensors and transmitted by things connected to the IoT. The data generated by things must be leveraged into more useful information for decision-making. The data is normally stored in a database. The organisation and processing of the data into usable information allows people to make informed decisions and take appropriate actions.
- **People** – People is an important element in the IoT ecosystem. People interact as both producers and consumers with the objective of improving well-being by satisfying human needs and wants.
- **Process -** Processes play a significant role in how the other elements of things, data, and people interact with each other to deliver value in the connected world of the IoT. With the right process, information is delivered to the right person, at the right time, in the appropriate way. Processes are easing communications between people, things, and data.

1.2 IoT Definition

The IoT is defined as the network of things, with clear element identification, embedded with software intelligence, sensors, and ubiquitous connectivity to the Internet. IoT is empowered by four main elements: sensors to collect information, identifiers to identify the source of data, software to analyze the data, and Internet connectivity to communicate and enable notifications. Sensors may be physical (e.g., sensors capturing the temperature) or logical (e.g., embedded software measurements such as CPU utilization). IoT's ultimate goal is to create a better environment for humanity, where objects around us know what we like, what we want, and what we need and act accordingly without explicit instructions. In its simple form, IoT may be considered as a network of physical elements empowered by:

- Sensors: to collect information.
- Identifiers: to identify the source of data (e.g., sensors, devices).
- Software: to analyze data.

- Internet connectivity: to communicate and notify.

Putting it all together, IoT is the network of things, with clear element identification, embedded with software intelligence, sensors, and ubiquitous connectivity to the Internet. IoT enables things or objects to exchange information with the manufacturer, operator, and/or other connected devices utilizing the telecommunications infrastructure of the Internet.

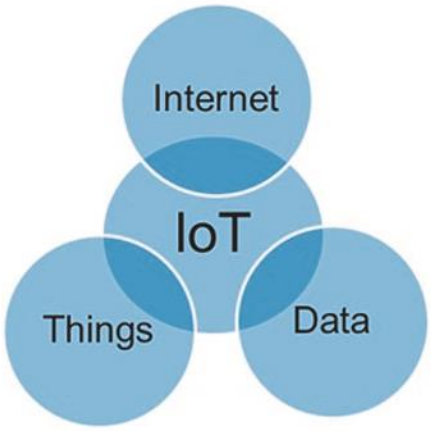


Fig.1: IoT definition

It allows physical objects to be sensed (to provide specific information) and controlled remotely across the Internet, thereby creating opportunities for more direct integration between the physical world and computer- based systems and resulting in improved efficiency, accuracy, and economic benefit. Each thing is uniquely identifiable through its embedded computing system and is able to interoperate within the existing Internet infrastructure

The main idea of IoT is to physically connect anything/everything (e.g., sensors, devices, machines, people, animals, trees) and processes over the Internet for monitoring and/or control functionality. Connections are not limited to information sites, they're actual and physical connections allowing users to reach "things" and take control when needed.

- **Background and More Complete IoT Definition**

Here we assume that the Internet is well known and bears no further definition. The question is what do we really mean by "Things2"? Well, things are actually "anything" and "everything" from appliances to buildings to cars to people to animals to trees to plants, etc. etc. Hence, IoT in its simplest form may be considered as the intersection of the Internet, things, and data as shown in **Fig. 1**. A more complete definition, we believe, should also include "Standards" and "Processes" allowing "Things" to be connected over the "Internet" to exchange "Data" using industry "Standards" that guarantee interoperability and enabling useful and mostly automated "Processes," as shown in Fig. 2.

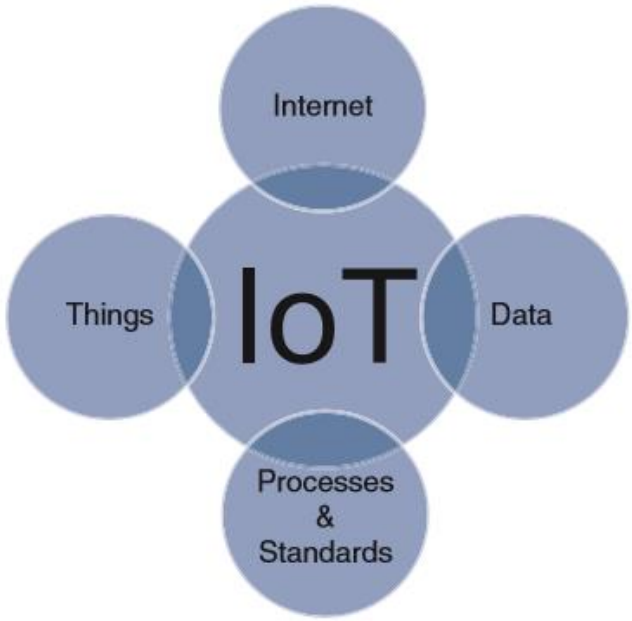


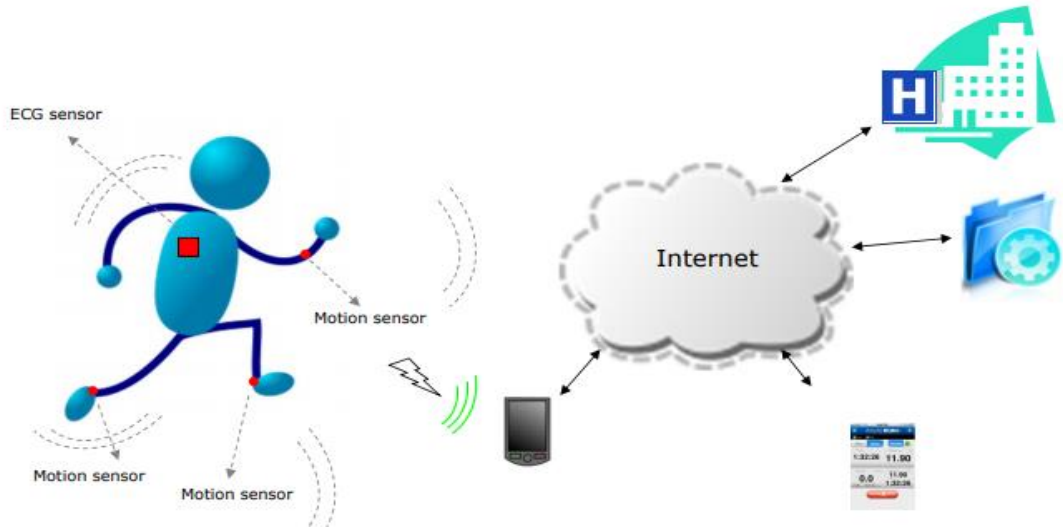
Fig. 2. IoT definition

Some companies (e.g., Cisco) refer to IoT as the IoE (Internet of Everything) with four key components: people, process, data, and Things. They correctly believe that today's Internet is the "Internet of People," i.e., today's Internet is mainly connecting applications that are used by people. People are taking action based on notifications from connected applications. IoT is envisioned to connect "things" where "things" (not people) will be taking action, when needed, by communicating with each other intelligently. IoE is then combining the Internet of People and the Internet of Things. In most of the recent literature, however, IoT refers to anything and everything (including people). With this in mind, we can state a more comprehensive definition

of IoT as follows: IoT is the network of things, with device identification, embedded intelligence, and sensing and acting capabilities, connecting people and things over the Internet.

A thing, in the Internet of Things, can be a person with a heart monitor implant, a farm animal with a biochip transponder, an automobile that has built-in sensors to alert the driver when tire pressure is low or any other natural or man-made object that can be assigned an IP address and provided with the ability to transfer data over a network. For instance, in the context of remote monitoring, a vending machine would alert the distributor when in short of a product. For example, think of it as your car with an IPv6 address that automatically connects to the internet to warn your gate to open a few minutes before you reach your home without any intervention on your side.

IoT to refer to a system involving **connected devices that gather data, connect with the Internet or local networks, generate analytics, and (in some cases) adapt behavior/responses based on the data/analytics in the network**.



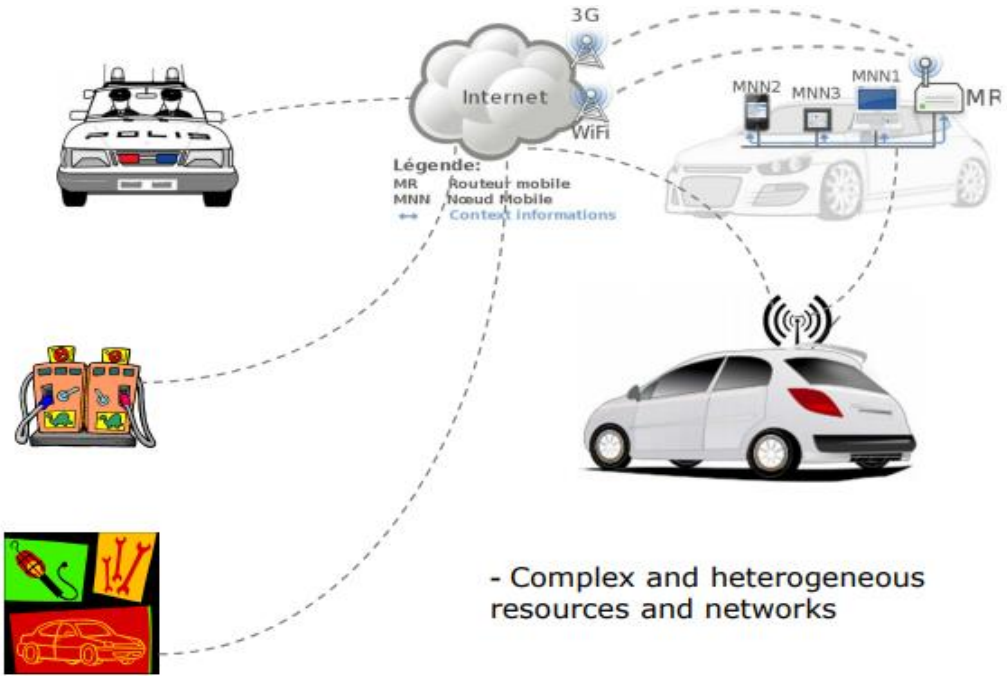
–‘‘a dynamic global network infrastructure with self-configuring capabilities based on standard and interoperable communication protocols where physical and virtual ’Things’ have identities, physical attributes, and virtual personalities and use intelligent interfaces, and are seamlessly integrated into the information network” (Kranenburg, 2008).

– ‘‘3A concept: anytime, anywhere and any media, resulting into sustained ratio between radio and man around 1:1” (Srivastava, 2006).

– ‘‘Things having identities and virtual personalities operating in smart spaces, using intelligent interfaces to connect and communicate within social, environmental, and user contexts” (Networked Enterprise & RFID & Micro & Nano-systems, 2008). The semantic meaning of ‘‘Internet of Things” is presented as ‘‘a world-wide network of interconnected objects uniquely addressable, based on standard communication protocols”.

– ‘‘A global infrastructure for the information society enabling advanced services by interconnecting (physical and virtual) things based on, existing and evolving, interoperable information and communication technologies” (ITU work on Internet of things, 2015).

In a nutshell, Internet of Things is defined as a “proposed development of the Internet in which everyday objects have network connectivity, allowing them to send and receive data.”



Wikipedia defines IoT as the internetworking of physical devices (also called smart devices and connected devices), vehicles, buildings and anything else embedded with electronics, sensors, actuators, software and network connectivity, which further enables these objects to collect and exchange data. In simple words, the Internet of Things refers to the vast world of different interconnected devices that have embedded sensors, which are capable of providing data and also being controlled through the Internet.

1.3 IoT Enabling Technologies

IoT is enabled by several technologies including Wireless Sensor Networks, Cloud Computing, Big Data Analytics, Embedded Systems, Security Protocols and architectures, Communication Protocols, Web Services, Mobile internet and semantic search engines.

1) **Wireless Sensor Network(WSN):** Comprises of distributed devices with sensors which are used work together to monitor the environmental and physical conditions.
2) **Cloud Computing:** Services are offered to users in different forms.

- **Infrastructure-as-a-service(IaaS):**provides users the ability to provision computing and storage resources. These resources are provided to the users as a virtual machine instances and virtual storage.
- **Platform-as-a-Service(PaaS):** provides users the ability to develop and deploy application in cloud using the development tools, APIs, software libraries and services provided by the cloud service provider.
- **Software-as-a-Service(SaaS):** provides the user a complete software application or the user interface to the application itself.

3) **Big Data Analytics:** Some examples of big data generated by IoT are
   - Sensor data generated by IoT systems.
   - Machine sensor data collected from sensors established in industrial and energy systems.
   - Health and fitness data generated IoT devices.
   - Data generated by IoT systems for location and tracking vehicles.
   - Data generated by retail inventory monitoring systems.
4) **Communication Protocols:** form the back-bone of IoT systems and enable network connectivity and coupling to applications.
   - Allow devices to exchange data over network.
   - Define the exchange formats, data encoding addressing schemes for device and routing of packets from source to destination.
   - It includes sequence control, flow control and retransmission of lost packets.
5) **Embedded Systems:** is a computer system that has computer hardware and software embedded to perform specific tasks. Embedded System range from low cost miniaturized devices such as digital watches to devices such as digital cameras, vending machines, appliances etc.,

1.4 Difference between M2M and IoT

Being whether at home, at work or in the street, we are constantly surrounded by IoT devices. The Internet of Things, probably the fastest growing phenomenon in the whole IT world, has already become a fully-fledged partner in our day-to-day activities, with smartphones, smart cars, smart

coffee machines and an endless list of other smart appliances gradually making their way into our lives and starting to drive our future. Therefore, getting the hang of the differences between the Internet of Things and Machine-To-Machine (M2M) communication, which is the underlying concept that gave rise to the IoT as we know it, seems to be an essential element in understanding what the whole IoT scene is all about nowadays.

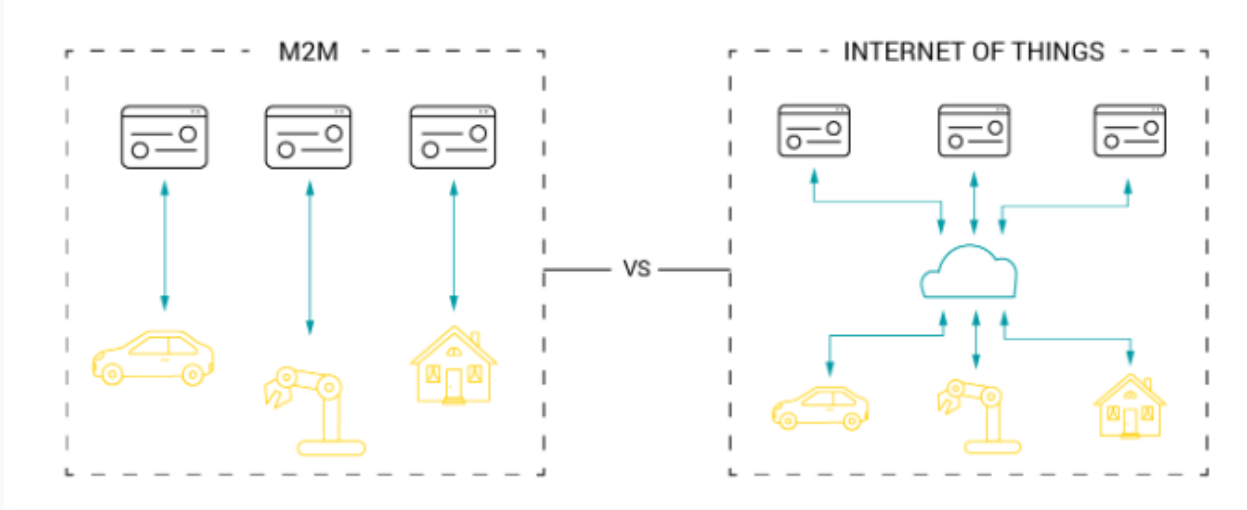
M2M has been used throughout the decades as the standard technology in telemetry even before the invention of the Internet itself, as it involved an interaction between two or more machines without human intervention. The idea of the Internet of Things, on the other hand, having evolved on the foundations laid down by M2M, aims at offering much more functionality. It uses Internet connectivity not only to enable communication between a fleet of the same kind of machines, but also to unite disparate devices and systems in an effort to marry different technology stacks and deliver interactive and fully integrated networks across varying environments.

M2M	**IoT**
M2M is about direct communication between machines using wired or wireless sysem.	IoT is based on IP networks based on standards to interconnect data from devices and a cloud platform or *middleware*
It support point to point communication	It supports cloud communication
Devices do not necessary rely on the Internet to communicate.	Devices rely on internet connection
M2M is mostly hardware-based technology	The IoT is both hardware and software-based technology
Machines normally communicate with a single machine at a time	Many users can access at one time over the internet
A device can be connected through mobile or other network	Data delivery depends on the internet protocol(IP) network

M2M, or machine-to-machine, is a direct communication between devices using wired or wireless communication channels. M2M refers to the interaction of two or more devices/machines that are connected to each other. These devices capture data and share with other connected devices, creating an intelligent network of things or systems. Devices could be sensors, actuators, embedded systems or other connected elements.

M2M technology could be present in our homes, offices, shopping malls and other places. Controlling electrical appliances like bulbs and fans using RF or Bluetooth from your smartphone is a simple example of M2M applications at home. Here, the electrical appliance and your smartphone are the two machines interacting with each other.

The Internet of Things (IoT) is the network of physical devices embedded with sensors, software and electronics, enabling these devices to communicate with each other and exchange data over a computer network. The things in the IoT refer to hardware devices uniquely identifiable through a network platform within the Internet infrastructure.



Interactivity: one of the main advantages of IoT in comparison to M2M is its capacity to interact by means of bidirectional communications. M2M, however, is based on unidirectional communications.

A real life example of this dissimilarity could be found in telemedicine. Let us imagine a solution that connects a sensor monitoring the heart rate of a patient to an external application which lets the doctor know the patient needs attention. Such kinds of solutions could easily be provided by the M2M technology.

On the other hand, if we take a sensor and integrate it with an interactive pillbox that would advise the patient to take the medicine and, moreover, would be able to send alerts to their family members' smartphones that the medicine has not been taken from the pillbox, it would definitely involve an IoT approach. To cut the story short, IoT could be viewed as M2M, but acting in a wider context.

1.5 Challenges of IoT

Though IoT usage is growing in different fields, there are a few challenges that require our attention and action so that it can be efficiently leveraged in areas where it is not yet being used. Let's have a look at some of these challenges.

1) **Data storage and analytics:** One of the challenges for developers of IoT applications is to clean, process and then interpret the large amounts of data gathered by different sensors. The proposed solution for this is to use wireless sensor networks. These networks share all the data that is gathered by sensor nodes, which is then sent to a distributed system for analytics. Another challenge is the storage of such large volumes of data.
2) **Platform fragmentation:** IoT also suffers from platform fragmentation (the inability to support a large number of platforms) and lack of common technical standards. Currently, a wide variety of IoT devices (in terms of both hardware and the differences in the software running on them) makes the task of developing different applications that work consistently across different inconsistent technical systems, difficult. Customers may be a bit hesitant to bet their IoT future on proprietary software or different hardware devices that use proprietary protocols in the fear that these may become obsolete or be difficult to customize.
3) **Privacy, autonomy and control:** Although IoT has immense potential to empower citizens by making governments transparent and by broadening information access, there are also serious threats to a citizen's privacy and the scope. Such concerns have led many to conclude that different Big Data infrastructures like the kind required for the Internet of Things and for data mining are incompatible with privacy.
4) **Security:** There have been many concerns raised that IoT is being developed rapidly without much thought being given to the profound security challenges associated with it and the different regulatory changes that might be necessary. When we talk of security concerns related to IoT, we refer to securing servers and workstations. The common measures like firewalling or security updates are unsuitable for much smaller IoT devices.
5) **Design:** The design and management of IoT must be sustainable and secure. The design of IoT devices must factor in uncertain futures with respect to their management, without risking physical failure. We cannot consider IoT devices to be successful without giving due

consideration to the interface's usability as well as the technology. The interfaces need to be not just user friendly but also better integrated.

6) **Complexity and unclear value propositions:** According to the feedback of several users, IoT solutions are either too complex or lack a clear use case for different end users. Experts also say that the IoT industry is currently heavily focused on gadgets, and is not making those gadgets relevant to particular business verticals. There are many who are just not able to pinpoint what value IoT offers them.

7) **Traditional governance structures:** There is a clash between IoT and the companies' traditional governance structures, as IoT still presents both uncertainties and the lack of historical precedence. Definite processes are needed to capture the IoT opportunity. This will help to improve the organizational design processes and to test the new innovation management practices.

8) **A lack of standards and interoperable technologies**: The sheer number of vendors, technologies and protocols used by each class of smart devices inhibits interoperability. The lack of consensus on how to apply emerging standards and protocols to allow smart objects to connect and collaborate makes it difficult for organizations to integrate applications and devices that use different network technologies and operate on different networks. Further, organizations need to ensure that smart devices can interact and work with multiple services.

9) Data and information management issues: Routing, capturing, analyzing and using the insights generated by huge volumes of IoT data in timely and relevant ways is a huge challenge with traditional infrastructures. The sheer magnitude of the data collected will require sophisticated algorithms that can sift, analyze and deliver value from data. As more devices enter the market, more data silos are formed, creating a complex network of connections between isolated data sources. The lack of universal standards and protocols will make it even tougher for organizations to eliminate data silos.

1.6 Characteristics:

1) **Dynamic & Self Adapting:** IoT devices and systems may have the capability to dynamically adapt with the changing contexts and take actions based on their operating conditions, user's context or sensed environment. Eg: the surveillance system is adapting itself based

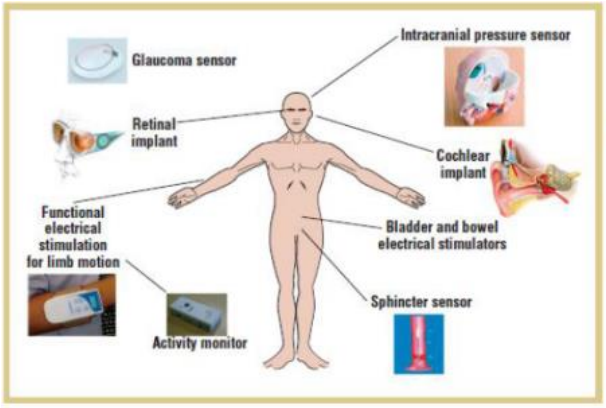
on context and changing conditions.

2) **Self-Configuring:** allowing a large number of devices to work together to provide certain functionality.
3) **Inter Operable Communication Protocols**: support a number of interoperable communication protocols and can communicate with other devices and also with infrastructure.
4) **Unique Identity**: Each IoT device has a unique identity and a unique identifier(IP address).
5) **Integrated into Information Network:** that allow them to communicate and exchange data with other devices and systems.

1.7 IoT Applications

Potential applications of the IoT are numerous and diverse, permeating into practically all areas of every-day life of individuals, enterprises, and society as a whole. The IoT application covers "smart" environments/spaces in domains such as: Transportation, Building, City, Lifestyle, Retail, Agriculture, Factory, Supply chain, Emergency, Healthcare, User interaction, Culture and tourism, Environment and Energy. Below are some of the IOT applications.

1.7.1 Healthcare Applications

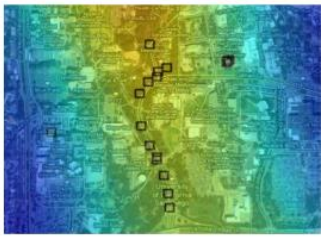


- Various sensors for various conditions
- Example ICP sensor: Short or long term monitoring of pressure in the brain cavity
- Implanted in the brain cavity and senses the increase of pressure
- Sensor and associated electronics encapsulated in safe and biodegradable material
- External RF reader powers the unit and receives the signal

1.7.2 Environmental Application

- Air quality monitoring project in UCSD CSE

- Environmental application
- Electrochemical **sensors**, **microcontroller** for data collection and transmission to an **Android** app
- **Actuation**: air quality is immediately reported, as well as retransmitted to a backend for larger-scale analysis

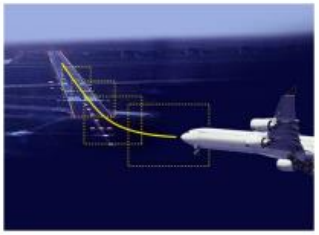


1.7.3 Transportation Applications

- Vehicle control: Airplanes, automobiles, autonomous vehicles
  – All kinds of sensors to provide accurate, redundant view of the world

– Several processors in cars (Engine control, break system, airbag deployment system, windshield wiper, door locks, entertainment system, etc.)

– Actuation is maintaining control of the vehicle



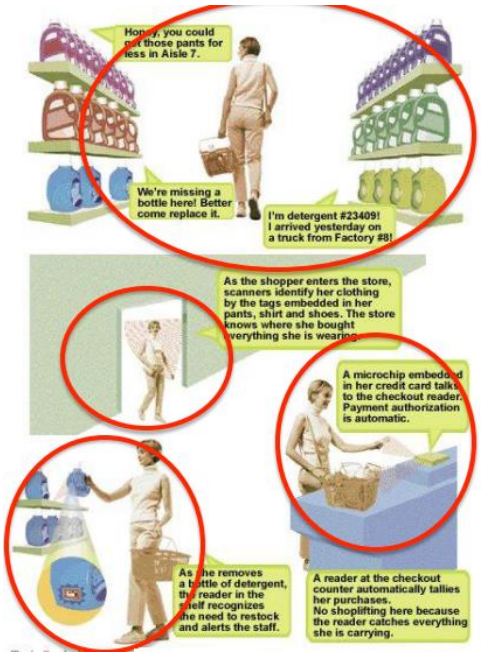
Example Transportation Scenarios

1. A network of sensors in a vehicle can interact with its surroundings to provide information – Local roads, weather and traffic conditions to the car driver – Adaptive drive systems to respond accordingly
2. Automatic activation of braking systems or speed control via fuel management systems.
   – Condition and event detection sensors can activate systems to maintain driver and passenger comfort and safety through the use of airbags and seatbelt pre-tensioning
3. Sensors for fatigue and mood monitoring based on driving conditions, driver behavior and facial indicators – Ensuring safe driving by activating warning systems or directly controlling the vehicle

1.7.4 Smart Home Applications

- Smart meters, hea1ng/cooling, mo1on/temperature/ lighting sensors, smart appliances, security, etc.

1.7.5 Future Shopping Application



- When entering the doors, scanners will identify the tags on her clothing.
- When shopping in the market, the goods will introduce themselves.
- When paying for the goods, the microchip of the credit card will communicate with checkout reader.
- When moving the goods, the reader will tell the staff to put a new one.

EXERICISES

1) What is the simple definition of IoT? What is the “more complete definition”? What’s the main difference?
2) IoT components were listed for the simple definition to include the intersection of the Internet, Things, and data. Process and standards were added to the complete definition. Why are process and standards important for the success of IoT?
3) What are the main four components that empower IoT? List the main function of each component.
4) What is IoT’s promise? What is IoT’s ultimate goal?

5) Cisco had estimated that the IoT would consist of almost 30 billion objects by 2020. Others have higher estimates. What was their logic?
6) In a table, list the 12 factors that are fueling IoT with a brief summary of each factor.
7) What are the top three challenges for IoT? Why are those challenges also considered as opportunities?
8) Why is operation technology (OT) under pressure to integrate with information technology (IT)?
9) Some companies use the term IoE instead of IoT. What is their logic?

UNIT 2: IOT ARCHITECTURE AND COMMUNICATION PROTOCOLS

2.1 OVERVIEW

IoT, Internet of Things is the term used to represent how “things” can be interconnected to the Internet to communicate and automate different processes. For example, think of a fridge that can detect the levels of products, and notify owner on the way back to home, to order them directly from the supermarket, and owner just need to drive through to pick up products. Other examples include, home automation, environmental monitoring, and so on.

2.2 LEARNING OUTCOMES

By the end of this unit, the students should be able to do the following:

- Identify the Components that forms part of IoT Architecture
- Determine the most appropriate IoT Devices and Sensors based on Case Studies.
- Setup the connections between the Devices and Sensors.
- Evaluate the appropriate protocol for communication between IoT 5.
- Analyze the communication protocols for IoT

2.3 IOT ARCHITECTURES

2.3.1 Introduction

IoT architecture consists of different layers of technologies supporting IoT. It serves to illustrate how various technologies relate to each other and to communicate the scalability, modularity and configuration of IoT deployments in different scenarios. There is no single consensus on architecture for IoT, which is agreed universally. Different architectures have been proposed by different researchers. IoT architecture varies from solution to solution, based on the type of solution which we intend to build.

2.3.2 Basic components of an IoT System

IoT as a technology majorly consists of four main components, over which an architecture is framed.

1) Sensors & Actuators
2) Devices
3) Gateway
4) Cloud

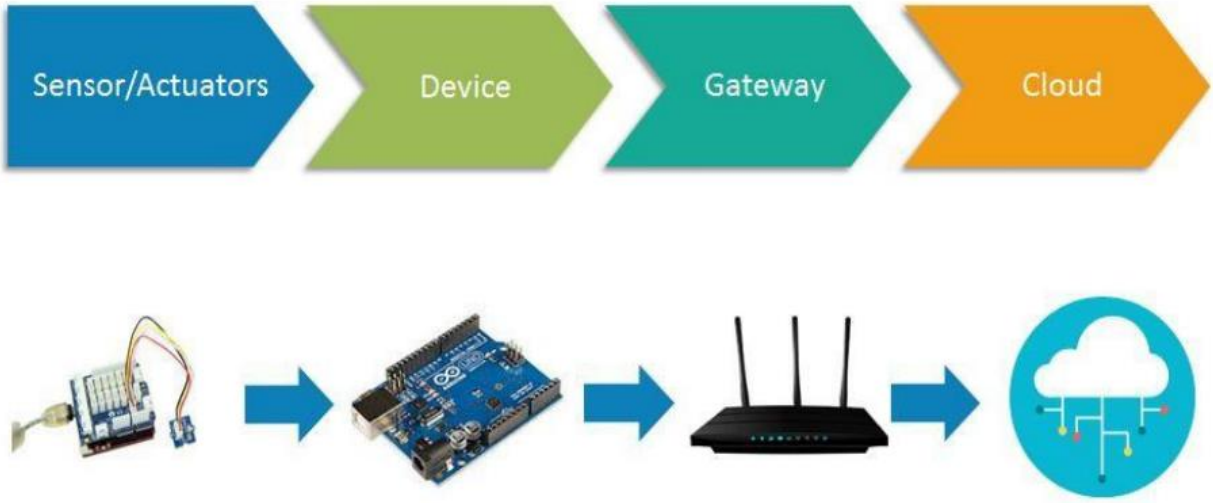


Fig. Stages of IoT architecture

2.3.2.1 Sensors/actuators

Sensors collect data from the environment or object under measurement and turn it into useful data. Think of the specialized structures in your cell phone that detect the directional pull of gravity and the phone's relative position to the —thing‖ we call the earth and convert it into data that your phone can use to orient the device. Actuators can also intervene to change the physical conditions that generate the data. An actuator might, for example, shut off a power supply, adjust an air flow valve, or move a robotic gripper in an assembly process.

The sensing/actuating stage covers everything from legacy industrial devices to robotic camera systems, water level detectors, air quality sensors, accelerometers, and heart rate monitors

2.3.2.2 DEVICE

A *device* includes hardware and software that directly interact with the world. Devices connect to a network to communicate with each other, or to centralized applications. Devices might be directly or indirectly connected to the internet.

2.3.2.3 GATEWAY

A *gateway* enables devices that are not directly connected to the Internet to reach cloud services. Although the term *gateway* has a specific function in networking, it is also used to describe a class of device that processes data on behalf of a group or cluster of devices.

A *gateway* manages traffic between networks that use different protocols. A gateway is responsible for ***protocol translation*** and other **interoperability** tasks. An IoT gateway device is sometimes employed to provide the connection and translation between devices and the cloud. Because some devices don't contain the network stack required for Internet connectivity, a gateway device acts as a proxy, receiving data from devices and packaging it for transmission over TCP/IP.
A dedicated gateway device might be a requirement if devices in the deployment:

- Don’t have routable connectivity to the Internet, for example, Bluetooth devices.
- Don’t have processing capability needed for transport-layer security (TLS), and as such can't communicate with Google APIs.
- Don't have the electrical power to perform required network transmission.

A gateway device might be used even when the participating devices are capable of communicating without one. In this scenario, the gateway adds value because it provides ***processing*** of the data across multiple devices before it is sent to the cloud. In that case, the direct inputs would be other devices, not individual sensors. The following tasks would likely be relegated to a gateway device:

- ***Condensing data to maximize*** the amount that can be sent to the cloud over a single link
- ***Storing data in a local database***, and then ***forwarding*** it on when the connection to cloud is intermittent
- ***Providing a real-time clock***, with a battery backup, used to provide a consistent timestamp for devices that can't manage timestamps well or keep them well synchronized
- *Performing* ***IPV6 to IPV4 translation***
- Ingesting and uploading other flat-file-based data from the local network that is relevant and associated with the IoT data
- Acting as a local cache for firmware updates

2.3.2.4 Cloud Platform

After your IoT project is up and running, many devices will be producing lots of data. You need an efficient, scalable, affordable way to both manage those devices and handle all that information and make it work for you. When it comes to storing, processing, and analyzing data, especially big data, it's hard to beat the cloud.

1) Data storage

Data from the physical world comes in various shapes and sizes. Cloud Platform offers a range of storage solutions from unstructured blobs of data, such as images or video streams, to structured entity storage of devices or transactions, and high-performance key-value databases for event and telemetry data.

2) Data Analytics

Performing analytics on data obtained through IoT sources is often the entire purpose of instrumentation for the physical world. After streaming data has been analyzed in a processing pipeline, it will begin to accumulate. Over time, this data provides a rich source of information for looking at trends, and can be combined with other data, including data from sources outside of your IoT devices.

The billions of sensor devices connected through the Internet generate a huge amount of digital data (so-called big data) that are generally stored in the digital domain using cloud computing services via the Internet. Advanced analytics helps generate meaningful information and actionable intelligence from these huge streams of data. This is an exciting opportunity for policy makers, governments, and industrial business owners to utilize analytics to predict, optimize, and

improve business and operations of public infrastructure.

Several big data processing tools, efficient databases, streaming analytics engines, and platforms have been developed to support IoT deployments in real-world applications. Popular analytic approaches include **deep learning, crowd analytics**, **anomaly detection engines, tracking algorithms, and pattern recognition and detection techniques**. In addition, **artificial intelligence models** have been developed for the public or users to interact with IoT technologies.

2.3.2 IoT Models

To be able to come up with a well-structured IoT architecture, the first step would be to review the existing IoT reference architectures proposed by different organizations as follows:

- The Industrial Internet Consortium has delivered the Industrial Internet Reference Architecture (IIRA), with a strong industry focus specifically on industrial IoT applications (Shi-Wan et al., 2017).
- The Internet of Things IoT-A EU initiative delivered a detailed architecture and model from the functional and information perspectives (Carrez, 2017).
- The Reference Architecture Model Industry 4.0 (RAMI 4.0) goes beyond IoT, adding manufacturing and logistics details. This is effectively a reference architecture for smart factories dedicated to IoT standards that started in Germany (Platform-i40.de, 2018).
- The IEEE P2413 project formed a working group dedicated to the IoT's architectural framework, highlighting protection, security, privacy and safety issues (Soo, 2016).

The common feature that can be derived from all the architectures mentioned above, is that they have well-defined boundaries for different activities clustered into layers, which shows how important aspects of the overall system operates and interacts among themselves. The layers can be adopted by different technologies, components, brands and so on. Conceptually, the layers can be identified as the following layers:

1) **Physical devices:** The "things" in the IoT
2) **The gateway:** A border module to link resource-limited devices
3) **Integration:** A protocol to allow communication between nodes and servers

4) **Applications:** Data processing and analytics, for example

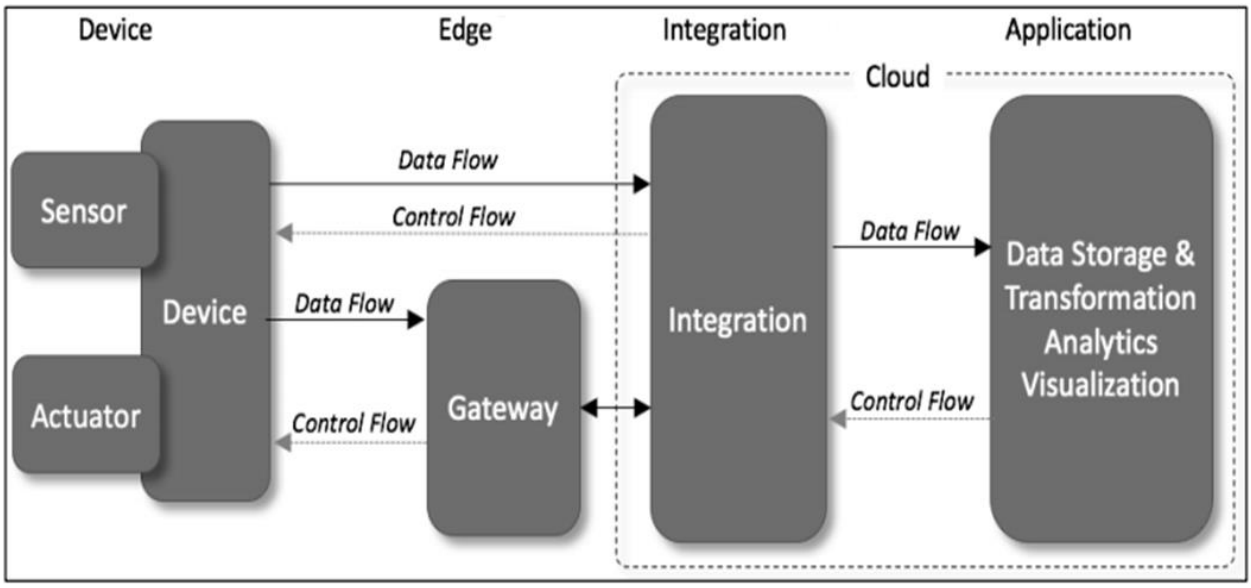


Fig.2.1 IoT Architecture (Walter, 2017)

Layer 1 – Physical Layer (Device Layer)

In this layer, devices such as sensors are used to capture parameters about the environment and to convert those into electrical signals. Also in this layer, actuators are being used for the process of automation to perform any mechanical processes. Communication between the devices and servers can happen in two ways.

1) Either each device communicates directly to the servers using protocols such as REST, MQTT, XMPP or AMQP (explained in next topic);
2) They can also communicate indirectly through a gateway. This will allow devices to just capture parameters and send them to the gateway through short-range communication protocols like Zigbee, Bluetooth and RF. The gateway would then combine the received the messages and transmit to the required server.

Layer 2 – Gateway Layer

In this layer, gathering of data from various devices is the main requirement. It performs translation of various messages received from heterogeneous devices and sends the leverage

data using protocols mentioned above. Apart from doing the receiving and sending of data, it can also provide services like data filtering, cleanup, aggregation and packet content inspection.

Layer 3 - Integration Layer

Most of the time this layer is hosted in the cloud. however, it could also be a dedicated server. This layer would be responsible for the collection of data from the gateways and storing in such way that they can be queried and processed. That is why the cloud is the preferred host as they also provided data redundancy. Examples are Apache Kafka, ThinkSpeed, Microsoft Azure (IoT).

Layer 4 - Application Layer

Finally, the Application Layer is responsible for the presenting the data collected, processed and labelled in such a way that it hides the complexities and is more inductive to users. In this layer, the data can be further enriched by applying machine learning algorithms and predictions can be made and presented to users. Presentation of final results can be through website or mobile apps. Examples of such platforms are IBM's Bluemix, Apache Spark, and so on.

2.3.3 General IoT Architecture

The next figure illustrates the form of general IoT architecture. For this intelligence and interconnection, IoT devices are equipped with embedded sensors, actuators, processors, and transceivers. IoT is not a single technology; rather it is an agglomeration of various technologies that work together.

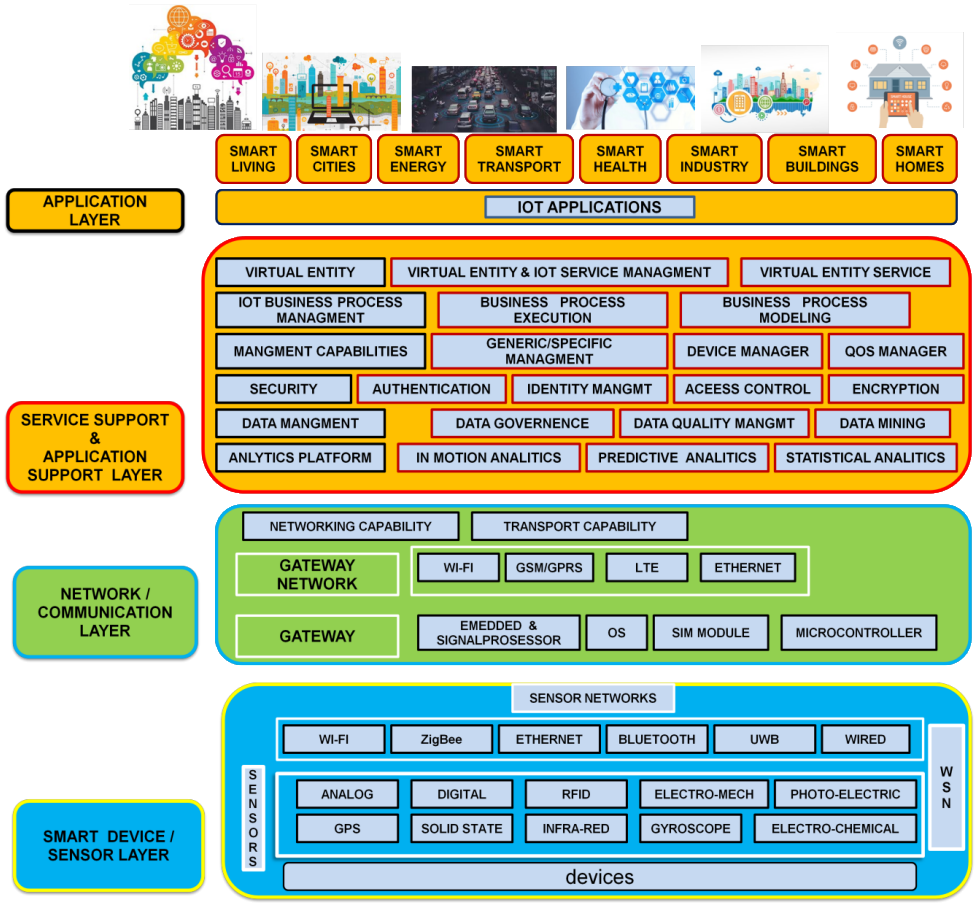


Fig2.2 General IoT Architecture

The functionality of each layer is described below:

2.3.3.1. Smart device/ Sensor Layer

The lowest layer is made up of smart objects integrated with sensors. The sensors enable the interconnection of the physical and digital worlds allowing real-time information to be collected and processed. The sensors have the capacity to take measurements such as temperature, air quality, speed, humidity, pressure, flow, movement and electricity etc. A sensor can measure the physical property and convert it into signal that can be understood by an instrument. Sensors are grouped according to their unique purpose such as environmental sensors, body sensors, home appliance sensors and vehicle telematics sensors, etc.

Most sensors require connectivity to the sensor gateways. This can be in the form of a Local Area Network (LAN) such as Ethernet and Wi-Fi connections or Personal Area Network (PAN) such as ZigBee, Bluetooth and Ultra Wideband (UWB). For sensors that do not require connectivity to

sensor aggregators, their connectivity to backend servers/applications can be provided using Wide Area Network (WAN) such as GSM, GPRS and LTE.

2.3.3.2 Gateways and Networks

Massive volume of data will be produced by these tiny sensors and this requires a robust and high performance wired or wireless network infrastructure as a transport medium. Current networks, often tied with very different protocols, have been used to support machine-to-machine (M2M) networks and their applications. With demand needed to serve a wider range of IoT services and applications such as high speed transactional services, context-aware applications, etc, multiple networks with various technologies and access protocols are needed to work with each other in a heterogeneous configuration. These networks can be in the form of a private, public or hybrid models and are built to support the communication requirements for latency, bandwidth or security.

2.3.3.3 Management Service Layer

The management service renders the processing of information possible through *analytics, security controls, process modeling and management of devices.*

IoT brings connection and interaction of objects and systems together providing information in the form of events or contextual data such as temperature of goods, current location and traffic data. Some of these events require filtering or routing to post- processing systems such as capturing of periodic sensory data, while others require response to the immediate situations such as reacting to emergencies on patient's health conditions. The rule engines support the formulation of decision logics and trigger interactive and automated processes to enable a more responsive IoT system.

In the **area of analytics**, various analytics tools are used to extract relevant information from massive amount of raw data and to be processed at a much faster rate. **Data management** is the ability to manage data information flow. With data management in the management service layer, information can be accessed, integrated and controlled.

Security must be enforced across the whole dimension of the IoT architecture right from the smart object layer all the way to the application layer. Security of the system prevents system hacking

and compromises by unauthorized personnel, thus reducing the possibility of risks.

2.3.3.4 Application Layer

The IoT application covers "smart" environments/spaces in domains such as: Transportation, Building, City, Lifestyle, Retail, Agriculture, Factory, Supply chain, Emergency, Healthcare, User interaction, Culture and tourism, Environment and Energy.

2.3.4 Three- and Five-Layer Architectures

The most basic architecture is a three-layer architecture as shown in

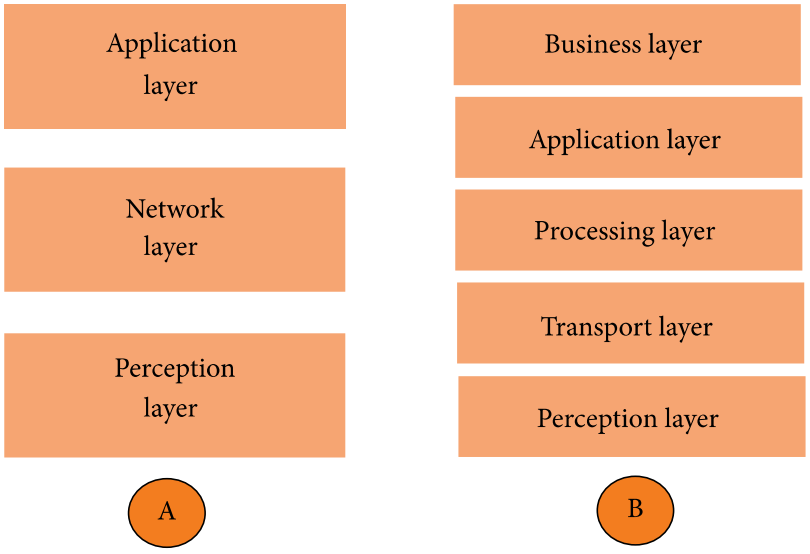


Fig2.3 Architecture of IoT (A: three layers) (B: five layers).

Three Layers was introduced in the early stages of research in this area. It has three layers, namely, the perception, network, and application layers.

(i) The *perception layer* is the physical layer, which has sensors for sensing and gathering information about the environment. It senses some physical parameters or identifies other smart objects in the environment.

(ii) The *network layer* is responsible for connecting to other smart things, network devices, and servers. Its features are also used for **transmitting and processing sensor data**.

(iii) The *application layer* is responsible for delivering application specific services to the user. It defines various applications in which the Internet of Things can be deployed, for example, smart homes, smart cities, and smart health.

The three-layer architecture defines the main idea of the Internet of Things, but it is not sufficient for research on IoT because research often focuses on finer aspects of the Internet of Things.

Another is the five-layer architecture, which additionally includes the processing and business layers. The five layers are perception, transport, processing, application, and business layers. The role of the perception and application layers is the same as the architecture with three layers. We outline the function of the remaining three layers.

(i) The *transport layer :* transfers the sensor data from the perception layer to the processing layer and vice versa through networks such as wireless, 3G, LAN, Bluetooth, RFID, and NFC.

(ii) The *processing layer:* is also known as the **middleware layer**. It stores, analyzes, and processes huge amounts of data that comes from the transport layer. It can manage and provide a diverse set of services to the lower layers. It employs many technologies such as databases, cloud computing, and big data processing modules.

(i) **The *business layer:*** manages the whole IoT system, including applications, business and profit models, and users' privacy.

2.3.5 Cloud and Fog Based Architectures.

These architectures are related to the nature of data processing. In some system architectures the data processing is done in a large centralized fashion by cloud computers. Such a cloud centric architecture keeps the cloud at the center, applications above it, and the network of smart things below it. ***Cloud computing*** is given primary because it provides great flexibility and scalability. It offers services such as the core infrastructure, platform, software, and storage. Developers can provide their storage tools, software tools, data mining, and machine learning tools, and visualization tools through the cloud.

Lately, there is a move towards another **system architecture**, namely, ***fog computing***, where the

sensors and network gateways do a part of the data processing and analytics. A fog architecture presents a layered approach which inserts ***monitoring, preprocessing, storage, and security layers*** between the physical and transport layers. The monitoring layer monitors power, resources, responses, and services. The preprocessing layer performs filtering, processing, and analytics of sensor data. The temporary storage layer provides storage functionalities such as data replication, distribution, and storage. Finally, the security layer performs encryption/decryption and ensures data integrity and privacy. Monitoring and preprocessing are done on the edge of the network before sending data to the cloud.

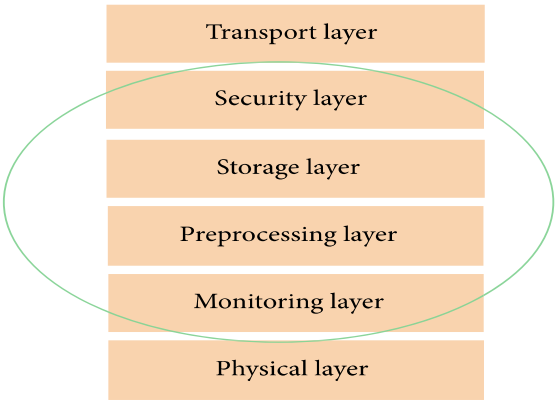


Fig2.4 Fog architecture of a smart IoT gateway.

Often the terms "fog computing" and "edge computing" are used interchangeably. The latter term predates the former and is construed to be more generic. ***Fog computing*** originally termed by Cisco refers to smart gateways and smart sensors, whereas *edge computing* is slightly more penetrative in nature. This paradigm envisions adding *smart data preprocessing capabilities* to physical devices such as motors, pumps, or lights. The aim is to do as much of preprocessing of data as possible in these devices, which are termed to be at the *edge* of the network.

The features of fog computing are as follows:

(1) **Low latency:** less time is required to access computing and storage resources on fog nodes (smart gateways).

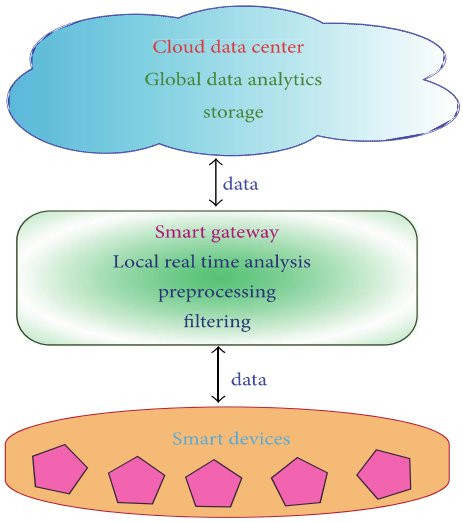


Fig2.5 Smart gateway for preprocessing.

Applications of fog computing include: (1) Smart vehicular networks: smart traffic lights are deployed as smart gateways to locally detect pedestrians and vehicles through sensors, calculate their distance and speed, and finally infer traffic conditions. This is used to warn oncoming vehicles. These sensors also interact with neighboring smart traffic lights to perform traffic management tasks. For example, if sensors detect an approaching ambulance, they can change the traffic lights to let the ambulance pass first and also inform other lights to do so. The data collected by these smart traffic lights are locally analyzed in real time to serve real time needs of traffic management.

Further, data from multiple gateways are combined and sent to the cloud for further global analysis of traffic in the city. (2) **Smart grid:** the smart electrical grid facilitates load balancing of energy on the basis of usage and availability. This is done in order to switch automatically to alternative sources of energy such as solar and wind power. This balancing can be done at the edge of the network using smart meters or microgrids connected by smart gateways. These gateways can analyze and process data. They can then project future energy demand, calculate the availability and price of power, and supply power from both conventional and alternative sources to consumers.

2.4. IoT Communication Protocols

2.4.1 Introduction

Internet of Things encompasses a large range of applications (Agriculture, Smart Home, Smart City, Smart Bus, Environment Monitoring, etc) that scale from one controlled device up to colossal amount of cross-platform distributions of different implanted technologies all connected together and in real-time through cloud systems. Making all those different devices talk to each other, and retrieving meaning information have led to the emergence of different communication protocols at each layer of the communication protocol stack.

2.4.2 IoT Networking Protocols

The actual TCP/IP networking model has 4 layers with protocols defined at each layer. However since IoT has its own specificity, different protocols have emerged at each layer to suit it. They are presented in Figure below

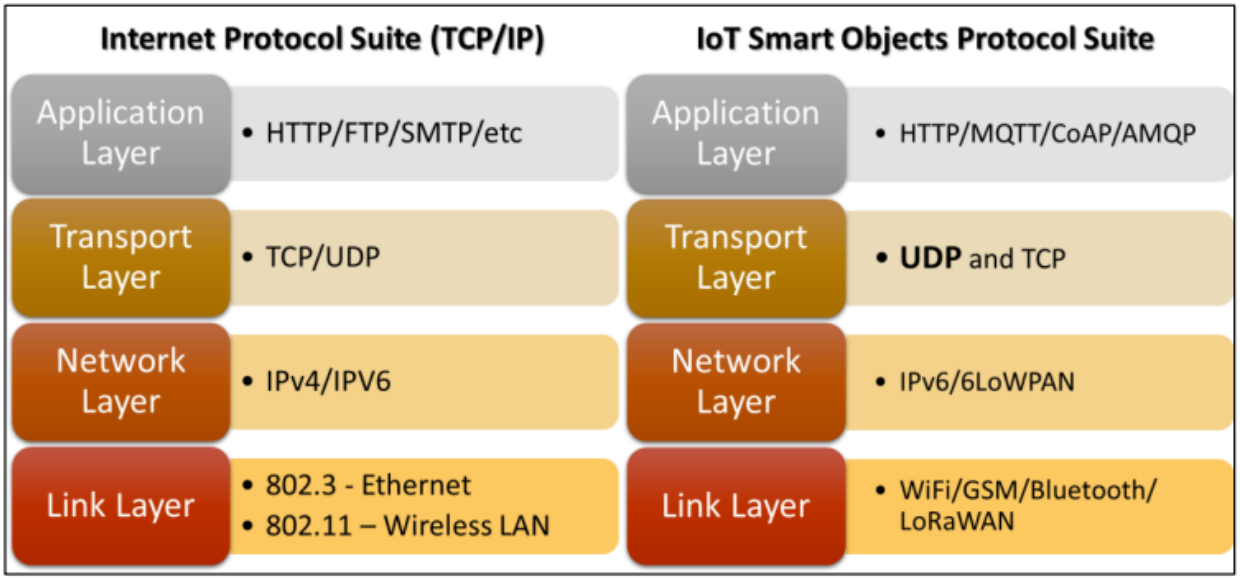


Fig2.6 – IoT and TCP/IP

2.4.2.1 Layer 1 (Link Layer)

–This layer concerns the infrastructure of the network. While current networking infrastructure uses mainly Wired or Wireless networking, IoT infrastructure uses a variety of technologies including:

1. **WiFi** – (Wireless Fidelity), latest version can cover up to 100m and can provide bandwidth up to about 1 Gbps.
2. **Bluetooth** – can have a data rate up to 3 Mbps and range from 10m and the maximum depends on manufacturer
3. **ZigBee** – works in the 2.4 GHz with a data rate of 250 Kbps and up to 100m or range. It also supports AES encryption.
4. **WiMax** – can cover large area up to 50 km (fix device) and 5-15 km (mobile device) and can provide a bandwidth up to 40 Mbps.
5. **Cellular (**2G/3G/4G) – provide worldwide coverage and varying bandwidth. However operational costs are higher.
6. **LoRaWAN-**LoRaWAN is a newly developed long-range wide-area network wireless technology designed for IoT applications with power saving, low cost, mobility, security, and bidirectional communication requirements. It is a low-power consumption optimized protocol designed for scalable wireless networks with millions of devices.

It supports redundant operation, location free, low cost, low power and energy harvesting technologies to support the future needs of IoT while enabling mobility and ease of use features This protocol is similar to Sigfox. Its data rates can vary from 0.3kbps to 50kbps, and it can be used within an urban or a suburban environment (2–5kms range in a crowded urban area). It was designed to serve as a standard for long range IoT protocols. It thus has features to support multitenancy, enable multiple applications, and include several different network domains.

7. **IEEE 802.11ah.**

The WiFi alliance has recently developed "**WiFi HaLow**" IoT communication protocol which is based on the IEEE 802.11ah standard. It consumes lower power than a traditional WiFi device and also has a longer range. This is why this protocol is suitable for Internet of Things applications.

The range of WiFi HaLow is nearly twice that of traditional WiFi.

Like other WiFi devices, devices supporting WiFi HaLow also support IP connectivity, which is important for IoT applications. Let us look at the specifications of the IEEE 802.11ah standard. This standard was developed to deal with wireless sensor network scenarios, where devices are energy constrained and require relatively long range communication. IEEE 802.11ah operates in the **sub-gigahertz band** (900 MHz). Because of the relatively lower frequency, the range is longer since higher frequency waves suffer from higher attenuation. We can extend the range (currently 1 km) by lowering the frequency further; however, the data rate will also be lower and thus the tradeoff is not justified.

8. RFID

Radio frequency identification (RFID) is the wireless use of electromagnetic fields to identify objects. Usually you would install an active reader, or reading tags that contain a stored information mostly authentication replies. Experts call that an Active Reader Passive Tag (ARPT) system. Short range RFID is about 10cm, but long range can go up to 200m. What many do not know is that Léon Theremin invented the RFID as an espionage tool for the Soviet Union in 1945.

An Active Reader Active Tag (ARAT) system uses active tags awoken with an interrogator signal from the active reader. Bands RFID runs on: 120–150 kHz (10cm), 3.56 MHz (10cm-1m), 433 MHz (1-100m), 865-868 MHz (Europe), 902-928 MHz (North America) (1-12m).

Examples include animal identification, factory data collection, road tolls, and building access. RFID tag is also attached to an inventory such that its **production and manufacturing progress can be tracked through the assembly line**.

Low Power Wide-Area-Networks (LPWAN).

Let us now discuss a protocol for long range communication in power constrained devices. The LPWAN class of protocols is low bit- rate communication technologies for such IoT scenarios.

Let us now discuss some of the most common technologies in this area.

9. **Band IoT**-It is a technology made for a large number of devices that are energy constrained. It is thus necessary to reduce the bit rate. This protocol can be deployed with both the cellular

phone GSM and LTE spectra. The downlink speeds vary between 40 kbps (LTE M2) and 10 Mbps (LTE category 1).

10. Sigfox-It is one more protocol that uses narrow band communication (≈10 MHz). It uses free sections of the radio spectrum (ISM band) to transmit its data. Instead of 4G networks, Sigfox focuses on using very long waves. Thus, the range can increase to a 1000 kms. Because of this the energy for transmission is significantly lower (<0.1%) than contemporary cell phones.
Again the cost is bandwidth. It can only transmit 12 bytes per message, and a device is limited to 140 messages per day. This is reasonable for many kinds of applications: submarine applications, sending control (emergency) codes, geolocation, monitoring remote locations, and medical applications.

10. Weightless

It uses a differential **binary phase shift keying** based method to transmit narrow band signals. To avoid interference, the **protocol hops** across frequency bands (instead of using CSMA). It supports cryptographic encryption and mobility. Along with frequency hopping, two additional mechanisms are used to reduce collisions. The downlink service uses time division multiple access (TDMA). Some applications include smart meters, vehicle tracking, health monitoring, and industrial machine monitoring.
Weightless is another newly developed wireless technology for the IoT MAC layer that is provided by the Weightless special interest group (SIG) - a non-profit global organization. Two standards can be used: Weightless-N and Weightless-W. Weightless-N was the first standard developed to support IoT requirements using TDMA with frequency hopping to minimize the interference. It uses ultra-narrow bands in the sub-1GHz ISM frequency band. On the other hand, Weightless-W provides the same features, but uses television band frequencies

2.4.2.2 Layers 2 &3: Network and Transport Layer

These layers look at the communication protocols which includes routing (choosing optimal path) and providing reliability (retransmission). Some existing protocols are being used, while some new ones have also seen the day.

1. **IPv6-** is the upgraded version of IPv4, which could not cope with boom of ip addresses demand. IPv6 can provide 1 IP address per cm2. It also has other features such as encryption and simple header (less overhead).
2. **6LoWPAN**-is the acronym of IPv6 over Low power Wireless Personal Area Networks. “It is an adaption layer for IPv6 over IEEE802.15.4 links. This protocol operates only in the 2.4 GHz frequency range with 250 kbps transfer rate" (Postscapes.com, 2017).
3. **UDP**-(User Datagram Protocol) is used an as alternative to TCP (reliable delivery), that provides performance tuned for real-time applications. However UDP does not provide reliability; therefore this aspect has to be addressed in higher level protocols.
4. Others include QUIC, Aeron, DTLS, NanoIP, CCN and TSMP.

2.4.2.3 Layer 4 (Application Layer)

This concerns the data protocols, that is specific for the application it is meant for. The application layer interacts with an application program, which is the highest-level layer. The application layer is the layer, which is closest to the end-user. It means the application layer allows users to interact with other software application. In this layer, the predominant protocol is HTTP (HyperText Transfer Protocol), which is still used by a lot of IoT Systems. However, some new protocols have also been implemented.

1. **HTTP –** mainly designed for web browsers, it also important for IoT as it supports the REST API which is becoming the main mechanism for Web Applications and services to communicate. However HTTP comes with a lot of overhead.
2. **MQTT -** (Message Queuing Telemetry Transport) " MQTT is a publish/subscribe protocol that runs over TCP. It was developed by IBM primarily as a client/server protocol. The clients are publishers/subscribers and the server acts as a broker to which clients connect through TCP.

Clients can publish or subscribe to a topic. This communication takes place through the broker whose job is to coordinate subscriptions and also authenticate the client for security. MQTT is a lightweight protocol, which makes it suitable for IoT applications. But because of the fact that it runs over TCP, it cannot be used with all types of IoT applications. Moreover, it uses text for topic names, which increases its overhead.

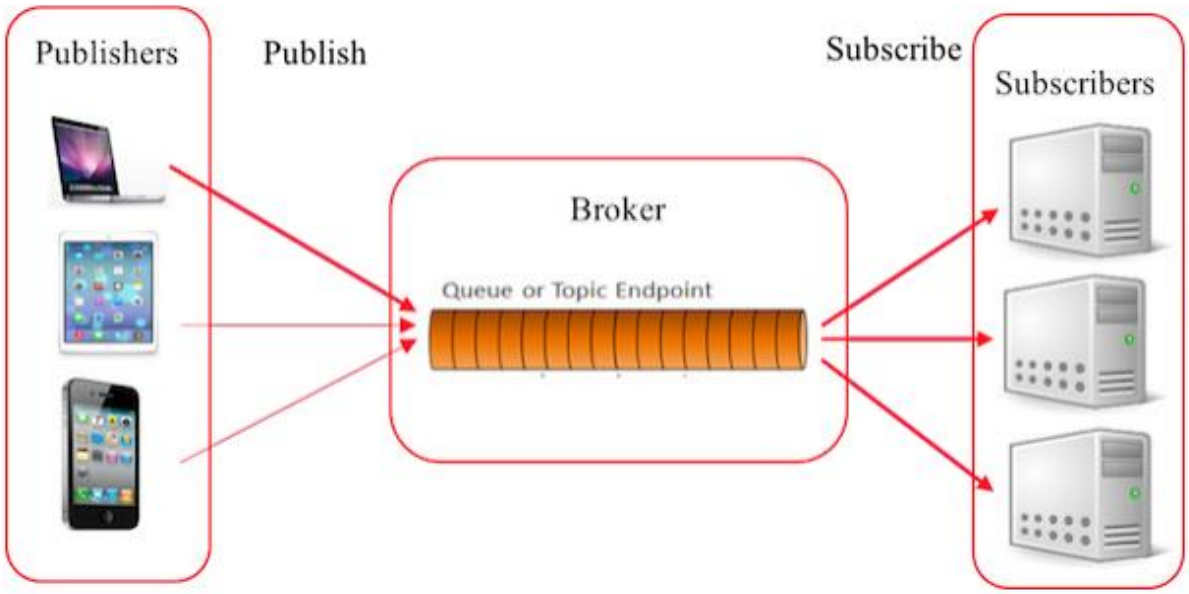


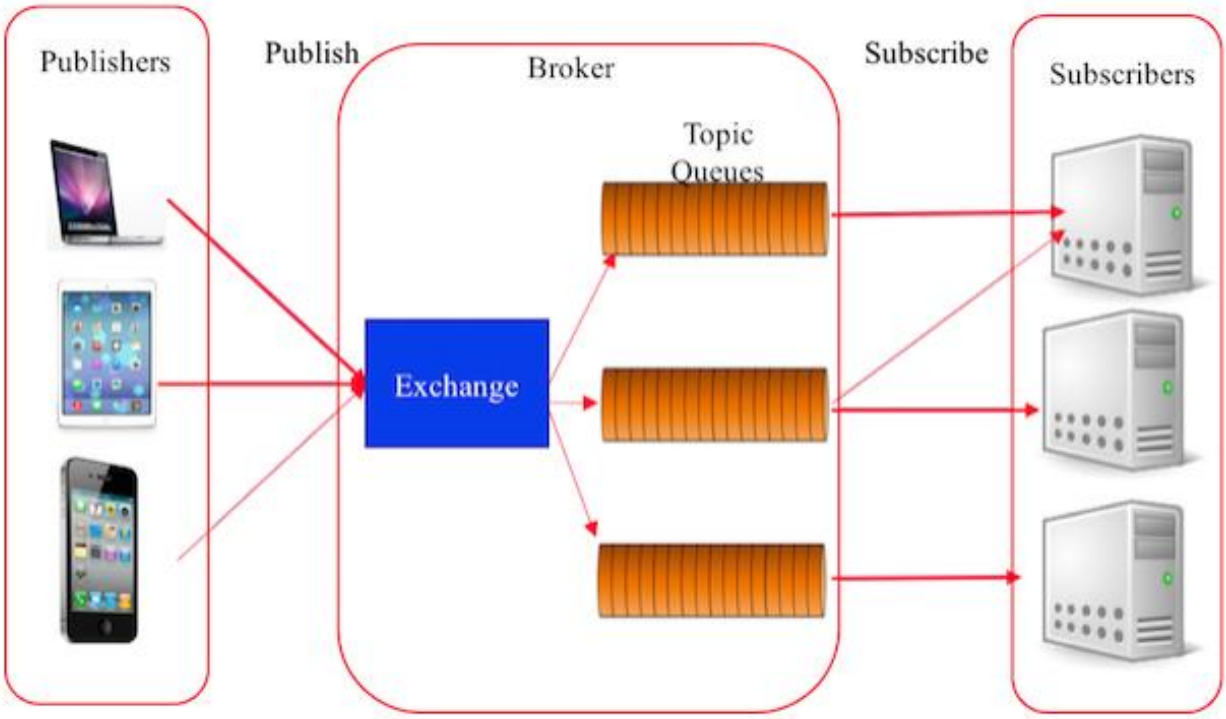
Fig. MQTT Architecture

Message queue telemetry transport (MQTT) is a 2013 standard from the Organization for the Advancement of Structured Information Standards (OASIS). It was introduced back in 1999 by IBM . It provides the connectivity between applications and users at one end, network and communications at the other end. It is a publish/subscribe architecture, as shown in Fig. above, where the system consists of three main components: publishers, subscribers, and a broker. In IoT, publishers are the lightweight sensors that connect to the broker to send their data and go back to sleep whenever possible. Subscribers are applications that are interested in a certain topic, or sensory data, so they connect to brokers to be informed whenever new data are received. The brokers classify sensory data in topics and send them to subscribers interested in those topics only.

3. **AMQP -** (Advanced Message Queuing Protocol) uses TCP/IP and publishes subscribe model and Point to Point.

Advanced message queuing protocol (AMQP) is another OASIS standard that was designed for the financial industry, runs over TCP, and uses publish/subscribe architecture, similar to MQTT. The main difference in these standards is that the broker is divided into two main components: exchange and queues, as shown in Fig. 6. The exchange component is responsible for receiving

publisher messages and distributing them to queues following predetermined roles. Subscribers connect to those queues, which basically represent the topics, and get the sensory data whenever they are available.



4. **COAP -** (Constrained Application Protocol) "is an application layer protocol that is intended for use in resource-constrained internet devices, such as WSN nodes. CoAP is designed to easily translate to HTTP for simplified integration with the web, while also meeting specialized requirements such as multicast support, very low overhead, and simplicity" (Postscapes.com, 2017).
5. Others include SMCP, XMPP, DDS, LLAP, Web Sockets and SOAP

UNIT 3: ASSEMBLY OF IoT DEVICE

3.0. LEARNING OUTCOMES

By the end of this unit, you should be able to do the following:

1. Describe and identify the Components that forms part of an IoT device
2. Identify different IoT platforms
3. Determine the most appropriate IoT Devices and Sensors based on Case Studies.
4. Setup the connections between the Devices and Sensors.

3.1 IoT Device

An IoT system may be treated as a system which can be physical, virtual, or a hybrid of the two, consisting of a collection of numerous active physical things, sensors, actuators, cloud services, specific IoT protocols, communication layers, users, developers, and enterprise layer.

A "**Thing**" in Internet of Things (IoT) can be any object that has a unique identifier and which can send/receive data (including user data) over a network (e.g., smart phone, smartTV, computer, refrigerator, car, etc.).

• IoT devices are connected to the Internet and send information about themselves or about their surroundings (e.g. information sensed by the connected sensors) over a network (to other devices or servers/storage) or allow actuation upon the physical entities/environment around them remotely.

The following are examples of an IoT device:

- A home automation device that allows remotely monitoring the status of appliances and controlling the appliances.
- An industrial machine which sends information abouts its operation and health monitoring data to a server.
- A car which sends information about its location to a cloud-based service.
- A wireless-enabled wearable device that measures data about a person such as the number of steps walked and sends the data to a cloud-based service.

3.2 Sensors

All IoT applications need to have one or more sensors to collect data from the environment. Sensors are essential components of smart objects. One of the most important aspects of the Internet of Things is *context awareness*, which is not possible without sensor technology. Schmidt and Van Laerhoven provide an overview of various types of sensors used for building smart applications.

3.2.1 Mobile Phone Based Sensors.

First of all, let us look at the mobile phone, which is ubiquitous and has many types of sensors embedded in it. In specific, the smartphone is a very handy and user friendly device that has a host of built in communication and data processing features. With the increasing popularity of smartphones among people, researchers are showing interest in building smart IoT solutions using smartphones because of the embedded sensors. Some of the sensors inside a modern smartphone are as follows:

(1) **The accelerometer** senses the motion and acceleration of a mobile phone. It typically

measures changes in velocity of the smartphone in three dimensions. The data patterns captured by the accelerometer can be used to detect physical activities of the user such as running, walking, and bicycling.

(2**) The gyroscope** detects the orientation of the phone very precisely. Orientation is measured using capacitive changes when a seismic mass moves in a particular direction.

(3) Th**e camera and microphone** are very powerful sensors since they capture visual and audio information, which can then be analyzed and processed to detect various types of contextual information. For example, we can infer a user's current environment and the interactions that she is having. To make sense of the audio data, technologies such as voice recogni**tion and acoustic features can be exploited.**

(4) The magnetometer detects magnetic fields. This can be used as a digital compass and in applications to detect the presence of metals.

(5) **The GPS** (Global Positioning System) detects the location of the phone, which is one of the most important pieces of contextual information for smart applications.

The location is detected using the principle of trilateration. The distance is measured from three or more satellites (or mobile phone towers in the case of A-GPS) and coordinates are computed.

(6) The **light sensor** detects the intensity of ambient light. It can be used for setting the brightness of the screen and other applications in which some action is to be taken depending on the intensity of ambient light. For example, we can control the lights in a room.

(7) **The proximity sensor** uses an infrared (IR) LED, which emits IR rays. These rays bounce back when they strike some object. Based on the difference in time, we can calculate the distance. In this way, the distance to different objects from the phone can be measured. For example, we can use it to determine when the phone is close to the face while talking.

(8) Some smartphones such as Samsung's Galaxy S4 also have a **thermometer, barometer,** and **humidity** sensor to measure the temperature, atmospheric pressure, and humidity, respectively.

3.2.2 Medical Sensors.

The Internet of Things can be really beneficial for health care applications. We can use sensors, which can measure and monitor various medical parameters in the human body. These applications can aim at monitoring a patient's health when they are not in hospital or when they are alone.

Subsequently, they can provide real time feedback to the doctor, relatives, or the patient. McGrath and Scanaill have described in detail the different sensors that can be worn on the body for monitoring a person's health.

There are many wearable sensing devices available in the market. They are equipped with medical sensors that are capable of measuring different parameters such as the heart rate, pulse, blood pressure, body temperature, respiration rate, and blood glucose levels . These wearables include smart watches, wristbands, monitoring patches, and smart textiles.

Fig: Smart watches

Another novel IoT device, which has a lot of promise are monitoring patches that are pasted on the skin. Monitoring patches are like tattoos. They are stretchable and disposable and are very cheap. These patches are supposed to be worn by the patient for a few days to monitor a vital health parameter continuously. All the electronic components are embedded in these rubbery structures. They can even transmit the sensed data wirelessly. Just like a tattoo, these patches can be applied on the skin. One of the most common applications of such patches is to monitor blood pressure.

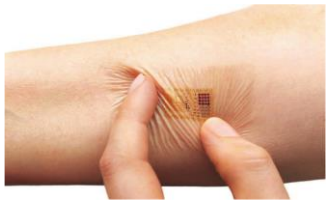


Figure 2: Embedded skin patches (source: MC10 Electronics).

3.2.3 Neural Sensors.

Today, it is possible to understand neural signals in the brain, infer the state of the brain, and train it for better attention and focus. This is known as neuro feedback. The technology used for reading brain signals is called EEG (Electroencephalography) or a brain computer interface. The neurons inside the brain communicate electronically and create an electric field, which can be measured

from outside in terms of frequencies. Brain waves can be categorized into alpha, beta, gamma, theta, and delta waves depending upon the frequency.

Figure: Brain sensing headband with embedded neuro-sensors (source: http://www.choosemuse.com/).

Based on the type of wave, it can be inferred whether the brain is calm or wandering in thoughts. This type of neurofeedback can be obtained in real time and can be used to train the brain to focus, pay better attention towards things, manage stress, and have better mental well-being.

3.2.4 Environmental and Chemical Sensors.

Environmental sensors are used to sense parameters in the physical environment such as temperature, humidity, pressure, water pollution, and air pollution. Parameters such as the temperature and pressure can be measured with a thermometer and barometer. Air quality can be measured with sensors, which sense the presence of gases and other particulate matter in the air.

Chemical sensors are used to detect chemical and bio- chemical substances. These sensors consist of a recognition element and a transducer. The electronic nose (e-nose) and electronic tongue (e-tongue) are technologies that can be used to sense chemicals on the basis of odor and taste, respectively. The e-nose and e-tongue consist of an array of chemical sensors coupled with advance pattern recognition software. The sensors inside the e-nose and e-tongue produce complex data, which is then analyzed through pattern recognition to identify the stimulus.

These sensors can be used in monitoring the pollution level in smart cities , keeping a check on food quality in smart kitchens, testing food, and agricultural products in supply chain applications.

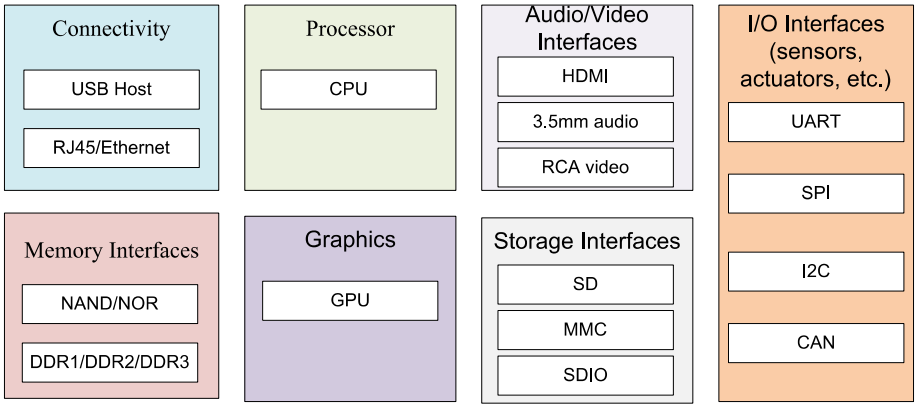
3.2.5 Radio Frequency Identification **(RFID).**

RFID is an identification technology in which an RFID tag (a small chip with an antenna) carries data, which is read by a RFID reader. The tag transmits the data stored in it via radio waves. It is similar to bar code technology. But unlike a traditional bar code, it does not require line of sight communication between the tag and the reader and can identify itself from a distance even without

a human operator. The range of RFID varies with the frequency. It can go up to hundreds of meters. RFID tags are of two types: **active and passive**. Active tags have a power source and passive tags do not have any power source. Passive tags draw power from the electromagnetic waves emitted by the reader and are thus cheap and have a long lifetime.

3.3 Components of an IoT Device

IoT system is comprised of a number of functional blocks to facilitate various utilities to the system such as, sensing, identification, actuation, communication, and management. The following Figure presents the components of an IoT device.



An IoT system is based on devices that provide sensing, actuation, control, and monitoring activities. IoT devices can exchange data with other connected devices and application, or collect data from other devices and process the data either locally or send the data to centralized servers or cloud based applications back-ends for processing the data, or perform some tasks locally and other tasks within IoT infrastructure based on temporal and space constraints (i.e. memory, processing capabilities, communicaion latencies, and speeds, and deadlines). An IoT device may consist of several interfaces for communications to other devices, both wired and wireless. These include (i) I/ O interfaces for sensors, (ii) interfaces for Internet connectivity, (iii) memory and storage interfaces, and (iv) audio/ video interfaces.

IoT devices can also be of varied types, for instance, wearable sensors, smart watches, LED lights, automobiles and industrial machines. Almost all IoT devices generate data in some form of the other which when processed by data analytics systems generate leads to useful information to guide further actions locally or remotely, For instance, sensor data generated by a soil moisture monitoring device in a garden, when processed can help in determining the optimum watering

schedules.

3.4 Device platforms

There is an incredible amount of diversity in the specific hardware available to you for building IoT applications. This diversity starts with the options for hardware platforms.

Common examples of platforms include single-board-computers such as the Beaglebone, and Raspberry Pi, as well as microcontroller platforms such as the Arduino series, boards from Particle, and the Adafruit Feather.

Each of these platforms lets you connect multiple types of sensor and actuator modules through a hardware interface.

3.5 .1 Hardware Interfaces

Most hardware interfaces are serial interfaces. Serial interfaces generally use multiple wires to control the flow and timing of binary information along the primary data wire. Each type of hardware interface defines a method of communicating between a peripheral and the central processor.

IoT hardware platforms use a number of common interfaces. Sensor and actuator modules can support one or more of these interfaces:

Universal Serial Bus (**USB**) is in common use for a wide array of plug-and-play capable devices.

General-purpose input/output pins **(GPIO)** are connected directly to the processor. As their name implies, these pins are provided by the manufacturer to enable custom usage scenarios that the manufacturer didn't design for. GPIO pins can be designed to carry digital or analog signals, and digital pins have only two states: HIGH or LOW. Digital GPIO can support Pulse Width Modulation (PWM). PWM lets you very quickly switch a power source on and off, with each "on" phase being a pulse of a particular duration, or *width*. The effect in the device can be a lower or higher power level.

For example, you can use PWM to change the brightness of an LED; the wider the "on" pulses, the brighter the LED glows. Analog pins might have access to an onboard analog-to-digital conversion (ADC) circuit.

An ADC periodically samples a continuous, analog waveform, such as an analog audio signal, giving each sample a digital value between zero and one, relative to the system voltage.

When you read the value of a digital I/O pin in code, the value must be either HIGH or LOW, where an analog input pin at any given moment could be any value in a range. The range depends on the resolution of the ADC. For example an 8-bit ADC can produce digital values from 0 to 255, while a 10-bit ADC can yield a wider range of values, from 0 to 1024. More values means higher resolution and thus a more faithful digital representation of any given analog signal. The ADC *samplingrate* determines the frequency range that an ADC can reproduce. A higher sampling rate results in a higher maximum frequency in the digital data. For example, an audio signal sampled at 44,100 Hz produces a digital audio file with a frequency response up to 22.5 kHz, ignoring typical filtering and other processing. The *bit precision* dictates the resolution of the amplitude of the signal.

Inter-Integrated Circuit (**I2C**) serial bus uses a protocol that enables multiple modules to be assigned a discrete address on the bus. I2C is sometimes pronounced "I two C", "I-I-C", or "I squared C".

Serial Peripheral Interface (**SPI**.) bus devices employ master-slave architecture, with a single master and full-duplex communication. SPI specifies four logic signals:

SCLK: Serial Clock, which is output from the master

MOSI: Master Output Slave Input, which is output from the master

MISO: Master Input Slave Output, which is output from a slave

SS: Slave Select, which is an active-low signal output from master

Universal Asynchronous Receiver/Transmitter (**UART)** devices translate data between serial and parallel forms at the point where the data is acted on by the processor. UART is required when serial data must be laid out in memory in a parallel fashion.

3.5.2 Microcontrollers

Microcontrollers were the first one to appear as an option for developers. They are small computing devices, easily connectable to hardware. The development tools were, however, an obstacle and made using them very complicated.

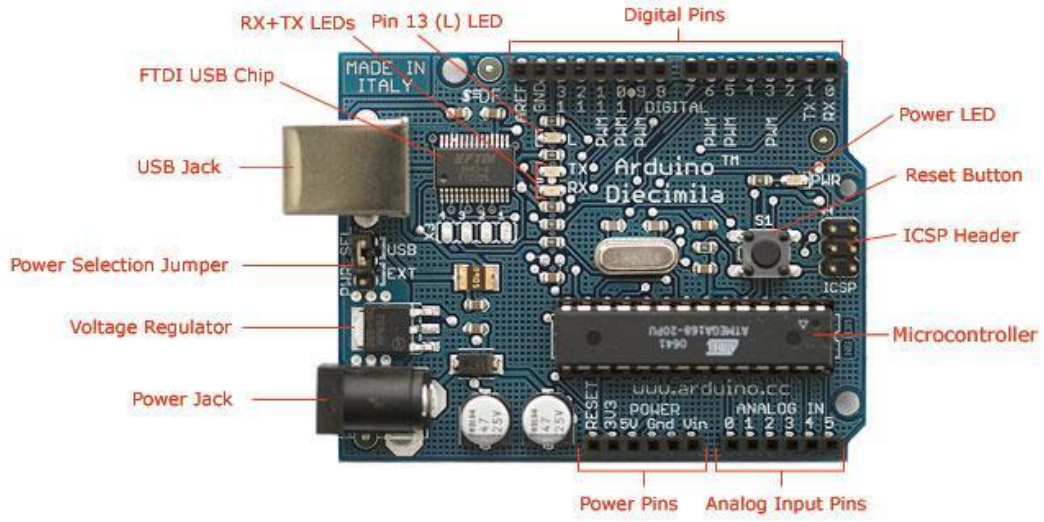
3.5.1. 1 Arduino

The first easy to use microcontroller was Arduino. It is a small programmable device, on which you can run simple, open source software called firmware. Shortly, the Arduino does not run an Operating System, what it runs is called Firmware. Basically, it runs only one program. Program written to the Arduino remain there until they are replaced with another program. Even when powered off, Arduino stores its software.

Arduino boards are the microcontrollers and microcontroller kit for building digital devices that can be sense and control objects in the physical and digital world. Arduino boards are furnished with a set of digital and analog input/output pins that may be the interfaced to various other circuits. Some Arduino boards include USB (Universal Serial Bus) to load programs from the personal computer.

● Inexpensive ● Quite easy to learn ● Flexible ● Good for sensing and controlling

- The programming language is based on **wiring** and in terms of syntax (almost) identical to C++.
- Arduino boards are based around Atmel processors (ATM168, ATM328).
- 8 bit controllers (new DUE board is first with 32 bit) 16 / 8 Mhz Approx. 32k of memory for code
- Run on 3.3, 5 (and up) Volts



Arduino compatible boards

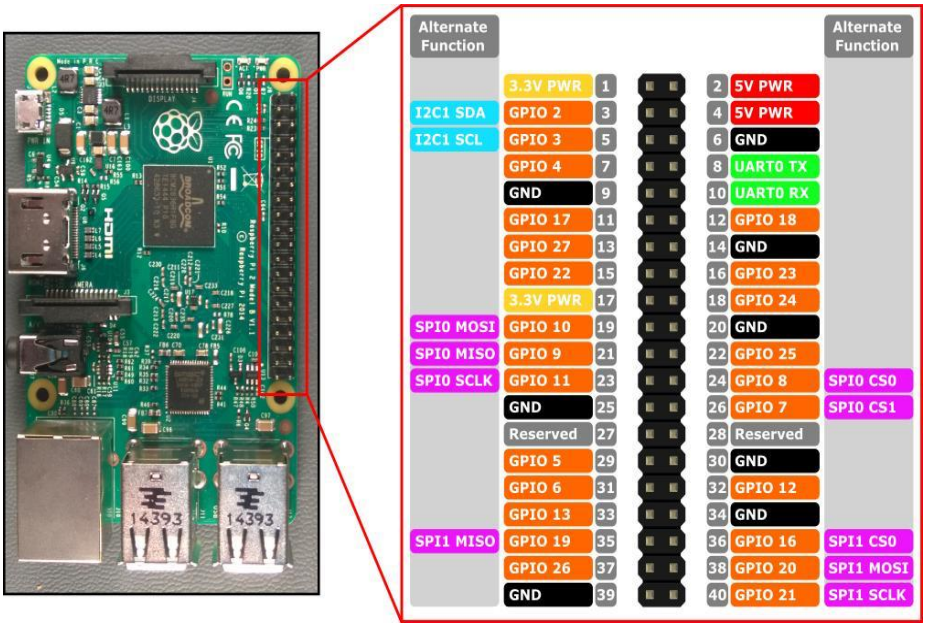
The ESP8266 is a low-cost Wi-Fi microchip with 32-bit microcontroller capability, standard digital peripheral interfaces. There are different types of ESP8266 boards are available for different needs. The primary goal of this board is to deal with the built-in Wifi through AT commands if used as device module, but you can 'program' using Arduino board however it also read and controls input/output, digital and analog.

3.5.1.2 Raspberry Pi

Raspberry Pi is a much popular device used in building IoT project. The recently launched Raspberry Pi 3 includes built-in WiFi and Bluetooth making the most compact and standalone computer. It provides a powerful environment to install a variety of programming packages such as Python, Node.js, LAMP stack, Java and much more. Using 40 GPIO pins, and four USB ports you can connect many peripherals and accessories to the Pi.

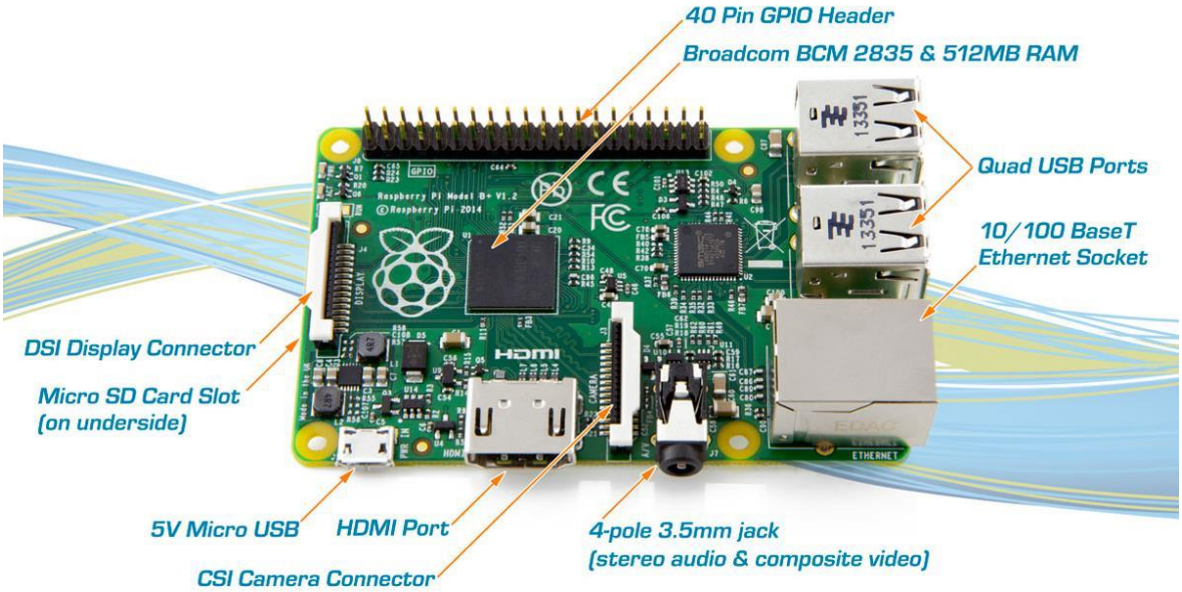
Conversely, since it is a computer, the Raspberry Pi runs an Operating System. That means that you can run multiple programs on it and you can run applications that use Internet services. It is actually a small computer, it runs Linux as operating system and has a full network system, solving thus the processing problem of the Arduino.

Pin Details

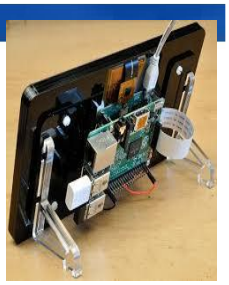


27x - GPIO pins
2x - SPI bus
1x - I2C bus
2x - 5V power pins
2x - 3.3V power pins
8x - Ground pins

Peripherals



Pi camera



DSI Display

Raspberry Pi Models

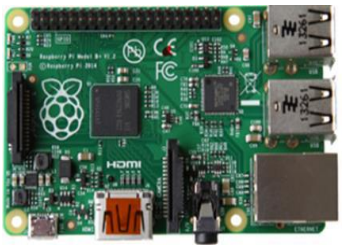
Model A

Model B

Pi 2, Model B

Model A+

Model B+



Pi 3, Model B

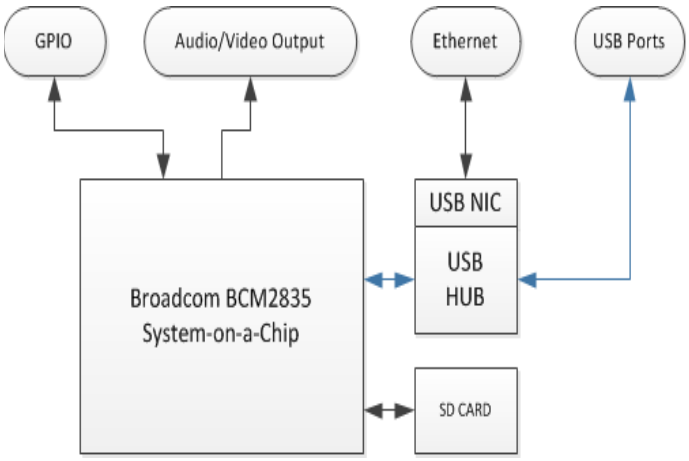
Comparison

Raspbrry pi	Model A+	Model B	Model B+	Model pi 2	Model pi 3
Chip :	Broadcom BCM2835			Broadcom BCM2836	Broadcom BCM2837
Processor:	ARMv6 Single core/700MHZ			ARMv7 quad core @900MHz	Quad core @1.2GHz
RAM:	256 MB SDRAM @ 400 MHz	512 MB SDRAM @ 400 MHz	512 MB SDRAM @ 400 MHz	1 GB SDRAM @ 400 MHz	1 GB SDRAM @ 400MHz
GPIO:	40	26	40	40	40
Max Power Draw/Voltage	1.8A@5V	1.8A@5V	1.8A@5V	1.8A@5V	2.5A@5V
USB 2.0:	1	2	4	4	4
Bluetooth/WiFi	NO	NO	NO	NO	YES

FEATURE	ARDUINO	RASPBERRYPI
Speed	16MHz	1.2GHz
Architecture	8 bit	64 bit
RAM	8KB	1GB
ROM	256KB	SD CARD(32GB)
Internet Connection	Not Easy	Easy
A/D	Yes	No
Programming Language	Arduino C/C++	No Limit
Multitasking	One Program at Time	Possible

System on Chip (SoC)

- A system on a chip (SoC) is a single microchip or integrated circuit (IC) that contains all the components needed for a system.
- SoCs are typically found on cell phones and embedded devices.
- For the Raspberry Pi, the SoC contains both an ARM processor for application processing and a Graphics Processing Unit (GPU) for video processing.



1) Hardware abstraction in software

An **operating system abstracts common computing resources such as memory and file I/O.** The OS also provides very low-level support for the different hardware interfaces. Generally these abstractions are not easy to use directly, and frequently the OS does not provide abstractions for the wide range of sensor and actuator modules you might encounter in building IoT solutions.

You can take advantage of **libraries that abstract hardware interfaces across platforms**. These libraries enable you to work with a device, such as a motion detector, in a more straightforward way. Using a library lets you focus on collecting the information the module provides to your application instead of on the low-level details of working directly with hardware.

Some libraries provide abstractions that represent peripherals in the form of lightweight drivers on top of the hardware interfaces. Examples of these libraries include the Johnny-Five JavaScript framework, MRAA, which supports multiple languages, the EMBD Go library, Arduino-wiring, and Firmata.

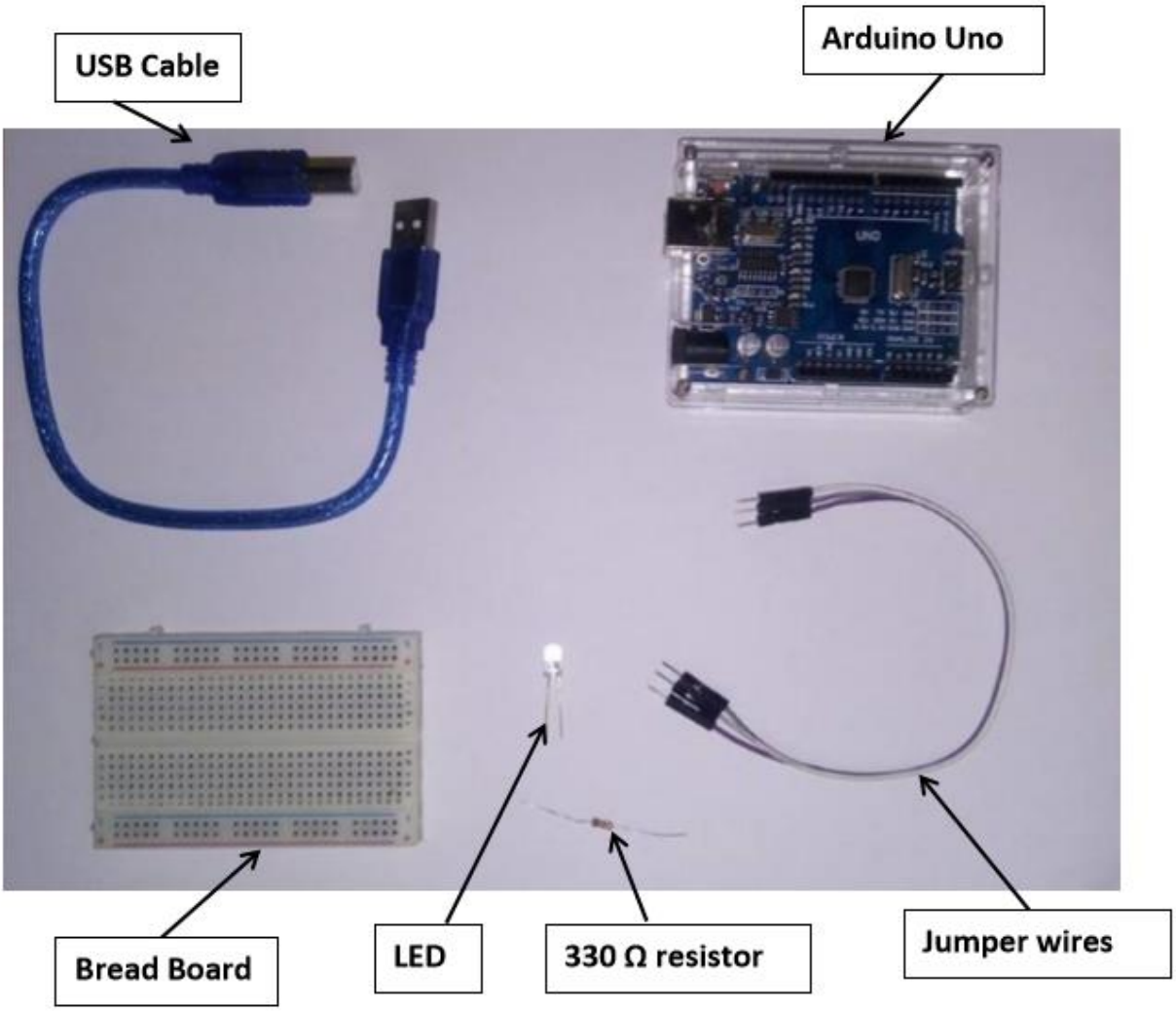
3.6 Case Study 1: Blinking Led

In this case study, we are going to test the hardware setup by writing a simple sketch (Arduino program) to blink a led and upload that program to the Arduino hardware.

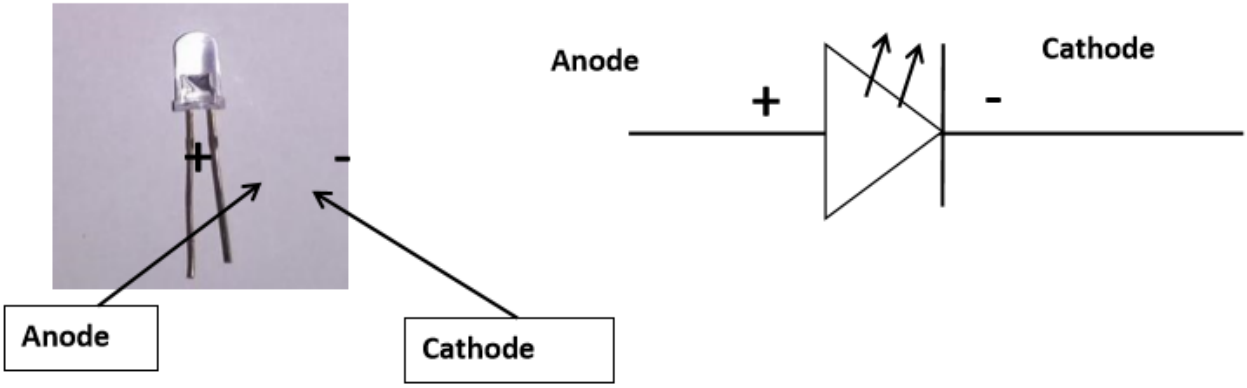
3.6.1.Pre-requirements:

i) Hardware

- Arduino Uno
- LEDs
- USB cable
- Jumper wires
- Resistors
- Breadboard



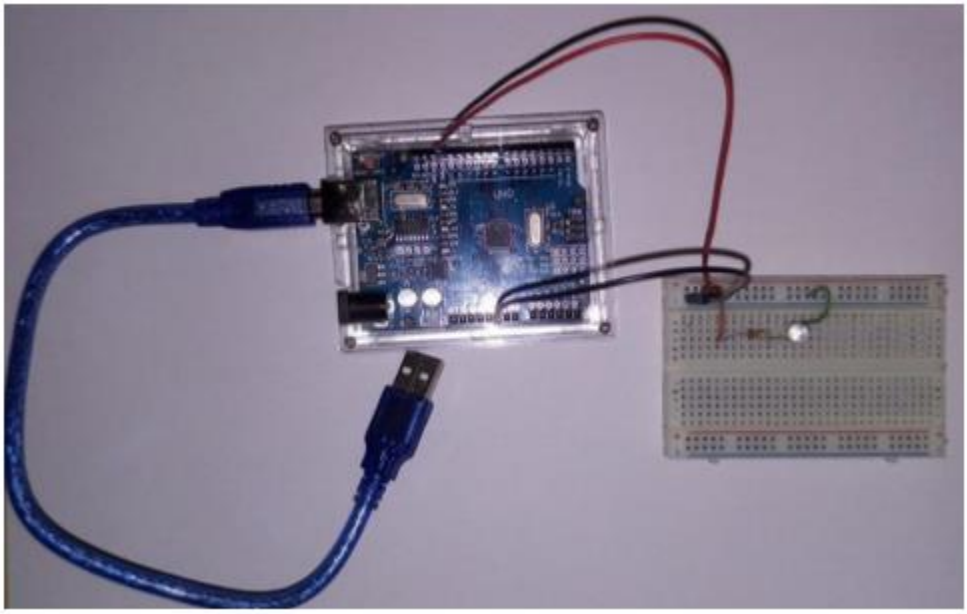
Note: All LEDs have a polarity that is a positive pin called Anode (the side with the longer pin) and a negative pin called Cathode (the side with the shorter pin) as shown in Figure below:

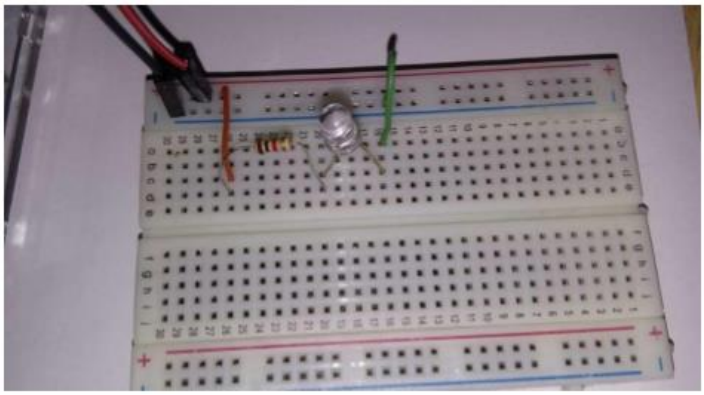


Connect the anode of the LED to the 330 ohm resistor (the side with the longer pin) and then use jumper wires to connect to the digital pin 13 (also called the led built-in) on the Arduino.

Connect the cathode of the LED to a jumper wire and connect the other part of the wire to the ground (GND) pin.

Then connect the USB port (side-B) to the Arduino board and the other port to your laptop/pc.





3.6.2 Software:

The hardware has been setup, but to achieve blinking on the LED, instructions have to be provided to the Arduino. This is where the Arduino IDE comes into play. If you don't have

installed arduino IDE, Arduino IDE which can be downloaded at https://www.arduino.cc/en/Main/Software. Now pen the Arduino IDE. This will load the IDE as shown in Figure below.

Before writing the sketch, it is important to make sure that the hardware is connected to the IDE so that the IDE can upload/update programmes on the hardware.

From the menu bar: Connection setup between laptop/pc to Arduino device:

- Select type of hardware: Tools -> Board -> Arduino/Genuino Uno.
- Select communication port: Tools -> Port -> Select port detected by your OS.

Note: If at this point, no port is detected, disconnect hardware and close IDE. Connect hardware (wait for it to be detected) and open IDE again.

Next step is to write the code to make the LED blink and upload it to the Arduino hardware.

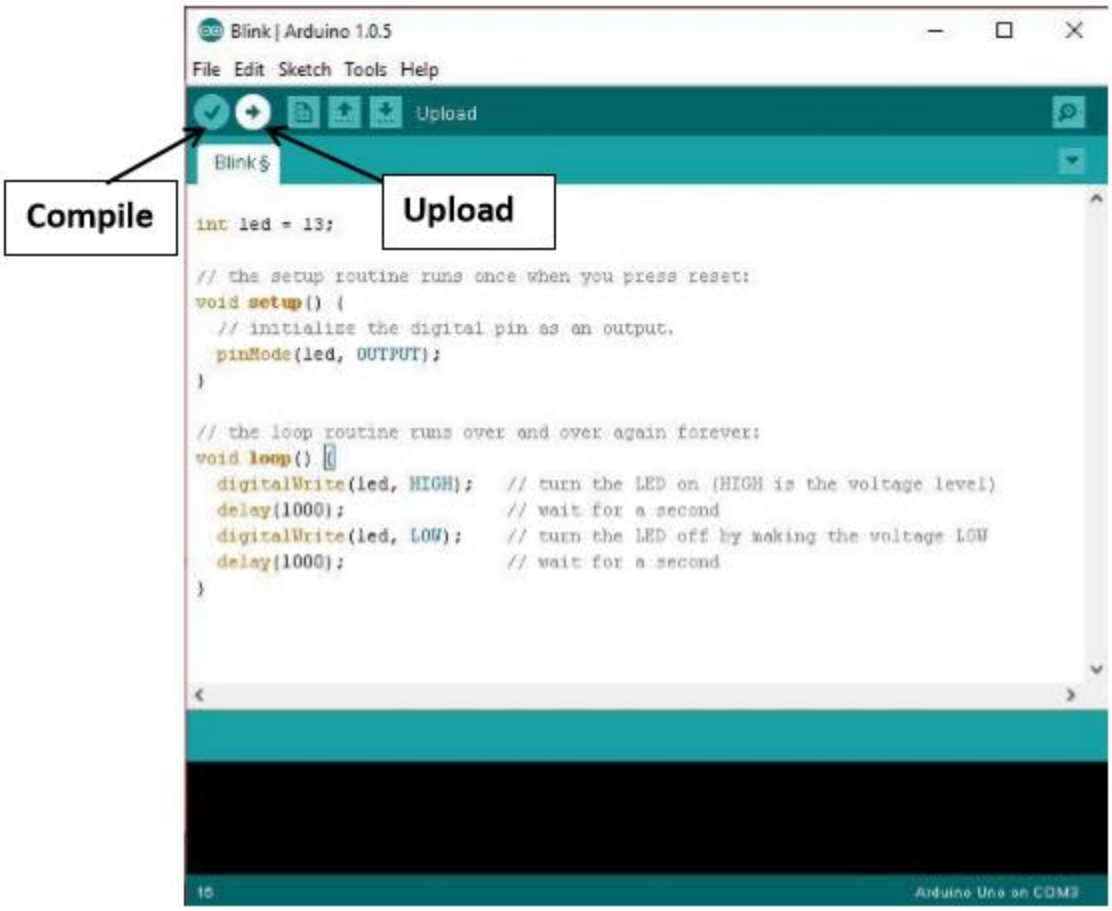
The Editor provides two predefined functions namely **setup ()** and **loop ().** Code used to initialize the Arduino will be placed in the setup () method since this function executes only

once. The loop () function as its name suggest will execute the code block inside it over and over.

To make the LED blink, we need to send power to it for a certain lapse of time (e.g., 1 second) so that it lights up and then stop the current to turn it off for a certain time (e.g., 0.5 seconds). Repeating this process will cause the blinking effect. This can be summarized according to the pseudo-code below:

```
BEGIN
INITIALISE
        SET digital pin (13) as OUTPUT
REPEAT
        Send HIGH voltage for 1 second to the pin used as output (to light LED)
        Send LOW voltage for 0.5 second to the pin used as output (to turn off LED)
END-REPEAT
END
```

The initialization part of the pseudo code will go in the setup() function whereas the repeating part will be written in the loop() function of the sketch as shown in the figure below



To compile and upload the code, click on the  and  respectively.

The Arduino Language Reference available at https://www.arduino.cc/reference/en/#functions explains the purpose of the different functions such as **pinMode(), digitalWrite(),** etc., and arguments such as **LED_BUILTIN, HIGH**, etc., used in the sketch above.

3.7 Case Study 2: Getting Temperature Values on Demand (Button Press)

In this case study, we are going to program an IoT device which will read a temperature sensor (LM35) and output the value of the temperature when the user presses on a push button.

3.7.1 Pre-requirements:

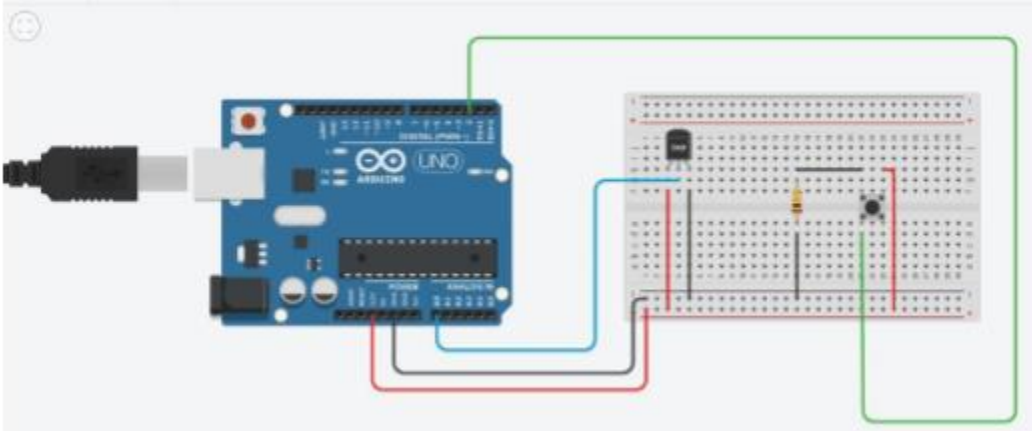
i. Hardware

- Arduino Uno
- Push button
- LM35 temperature sensor
- USB cable
- Jumper wires
- Resistor 10k Ohm
- Breadboard

ii. Software:

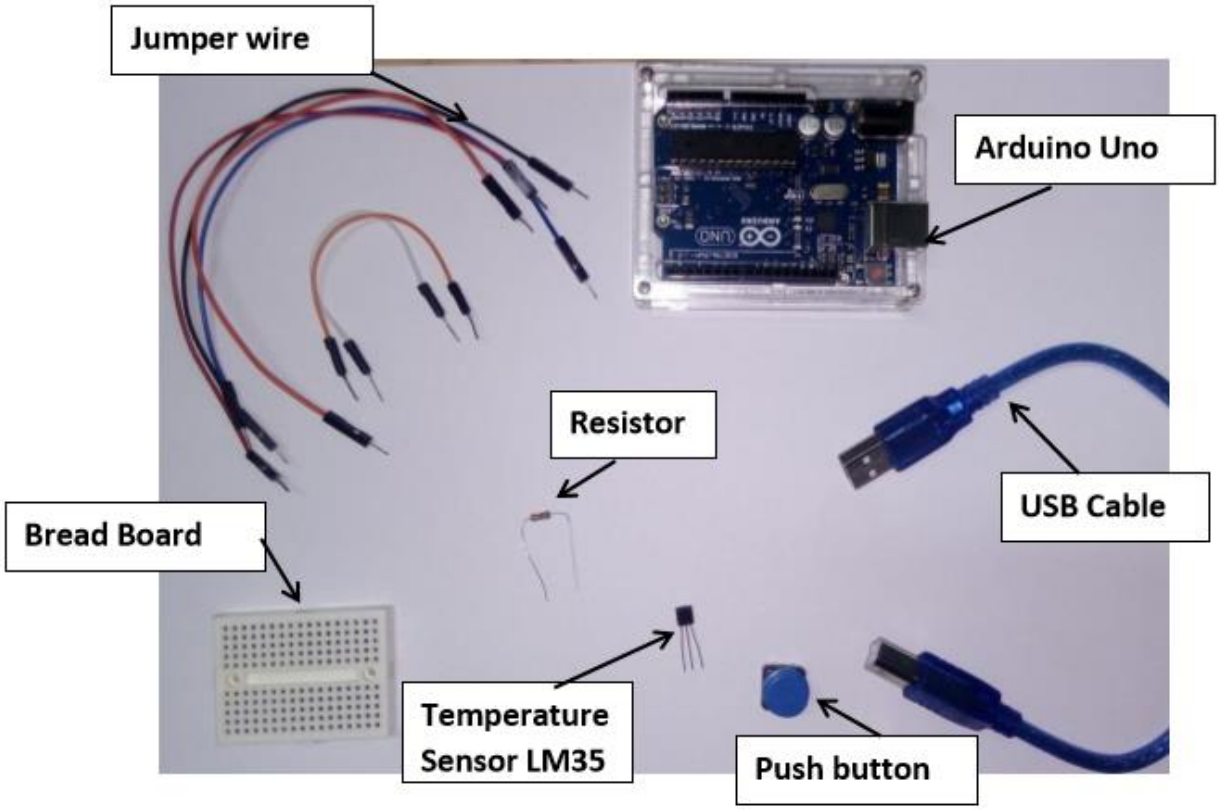
- Arduino IDE which can be downloaded at https://www.arduino.cc/en/Main/Software .

3.7.2 Designing circuit



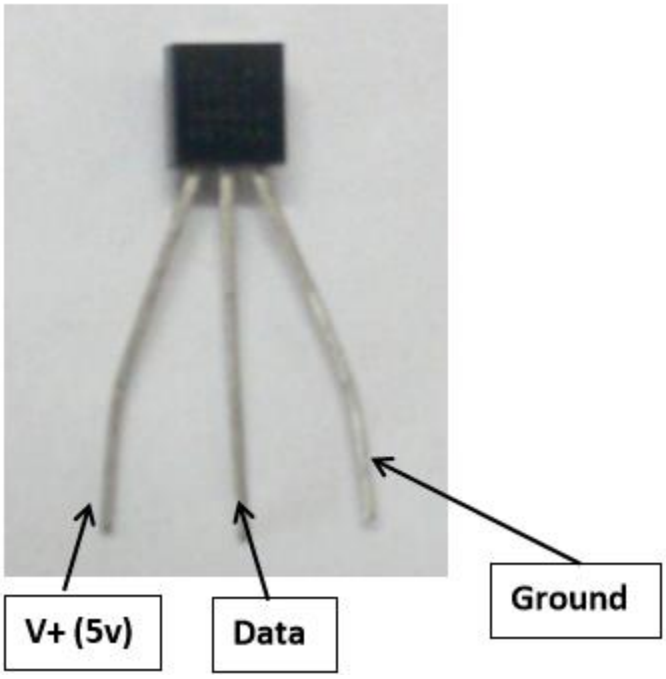
3.7.3 Setting up the IoT hardware

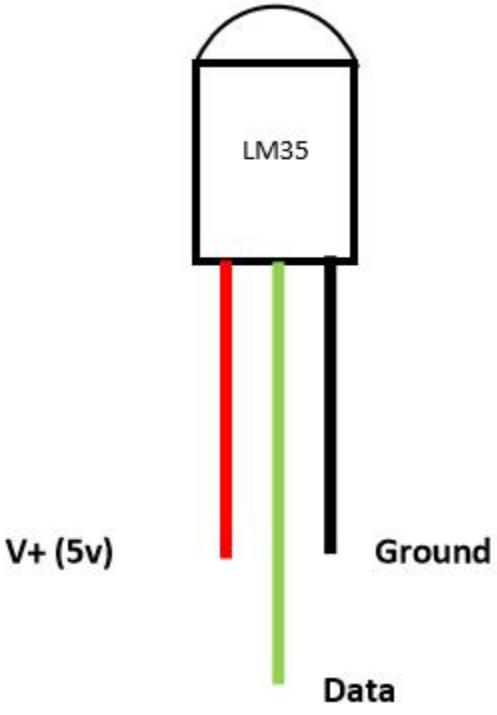
You will need the Arduino Uno, the USB A to B cable to connect the Arduino to your PC/Laptop, Temperature sensor LM35, Jumper cables, 10k Ohm resistor, a push button and a breadboard.



Factsheet about LM35 Temperature Sensor:

Temperature sensor pinout:





Facing the flat part of the sensor, the pins are labelled as shown in Figure above. The sensor is connected as follows:

- The positive pin is connected to the positive 5v;
- The negative is connected to the ground and the signal pin is connected to the analog pin A0 of the Arduino.

Additional details:

Push button (connection to Arduino through jumper wires):

The top left pin connects to the 10k Ohm resistor which will in turn connects to the ground pin. The top right pin connects to the 3.3 volt pin while the bottom left pin connects to the digital pin 2 of the Arduino.

LM35 Temperature sensor (connection to Arduino through jumper wires):

The anode (+ve pin) of the temperature sensor connects to the 3.3 volt pin of the Arduino, while the data pin connects to analog pin A0 and the ground pin of the sensor connects to the ground pin of the Arduino.

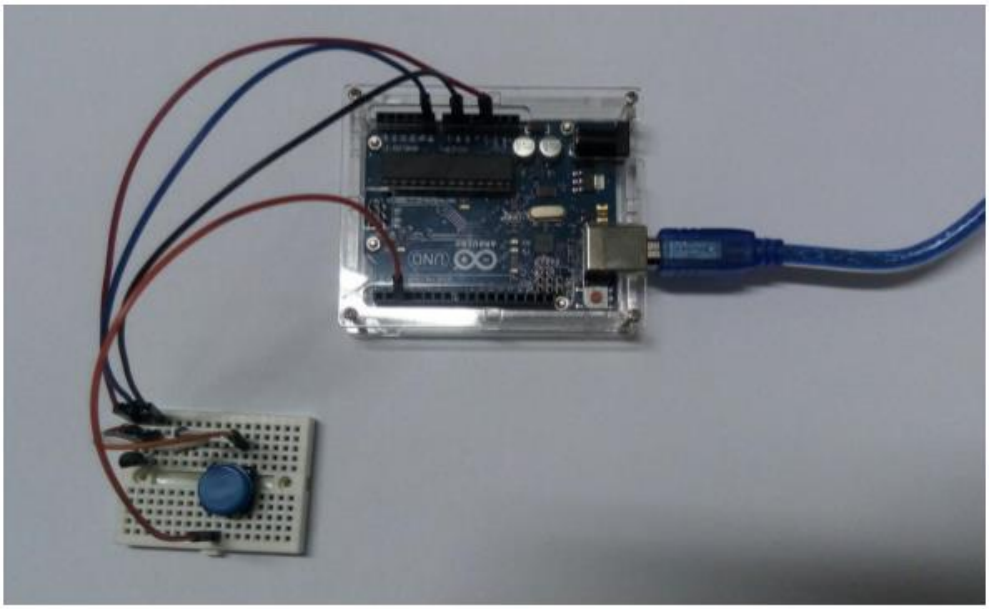


Fig. top view

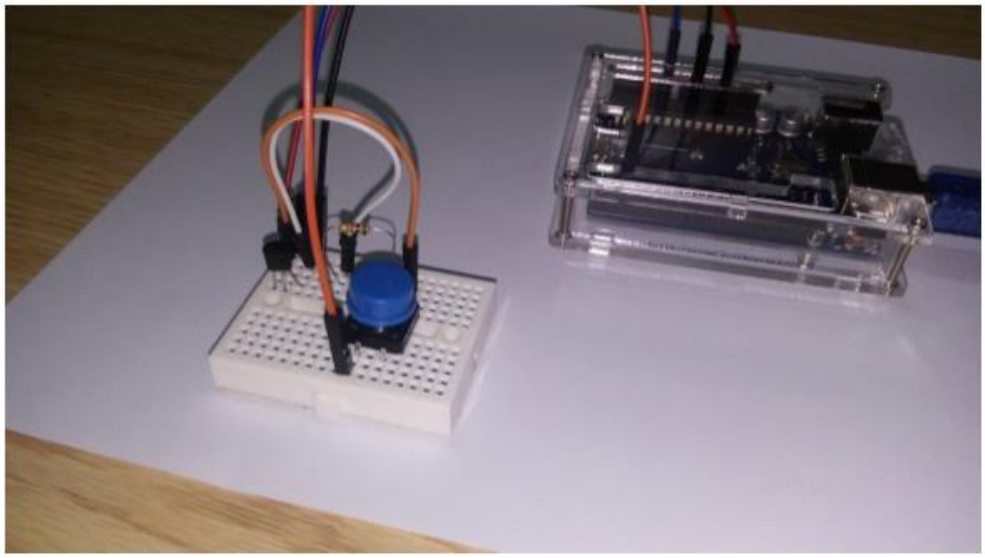


Fig. side view

Next step is to write the program to calculate and display the temperature value from the LM35 sensor when the push button is pushed. Note that the sensor only sends electrical signal to the Arduino. The sketch will have to interpret these values and return the equivalent in degrees Celsius. For that purpose, we are going to use a formula to convert the electrical values to temperature values.

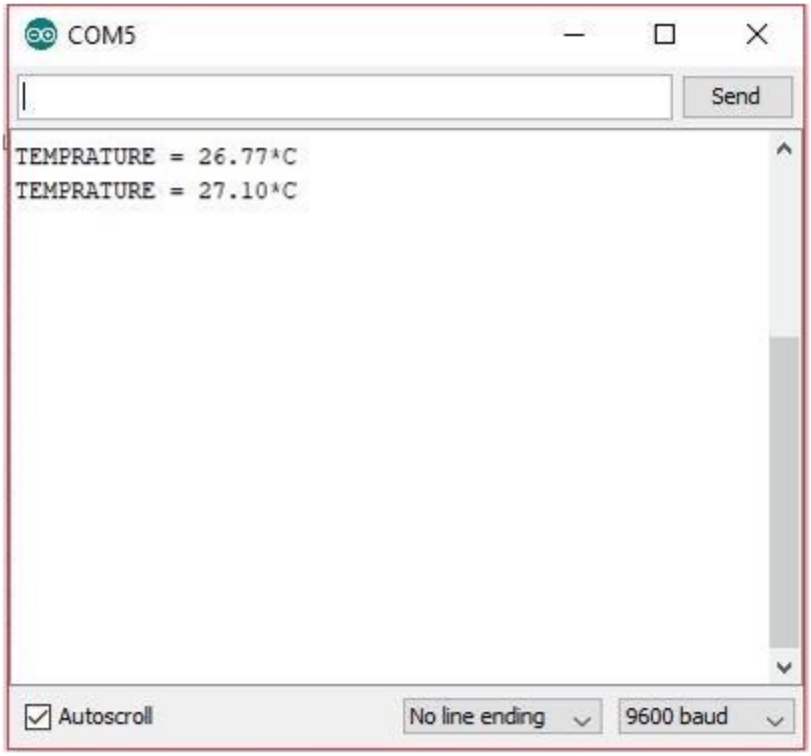
This can be summarized according to the pseudo code below.

```
BEGIN
INITIALISE
        SET digital pin2 as INPUT
REPEAT
        Read value from pin2
        Check if button is pushed.
        If True
                Apply formula to convert readings to degrees Celsius
                Send data to Serial port
                Display data on serial monitor
        Else
                Do nothing
        End if
END-REPEAT
END
```

The sketch resulting from the above pseudo code is shown below:

```
float resolution=3.3/1023;   // 3.3 is the supply volt  & 1023 is max analog read value
const int buttonPin = 2;     // the number of the pushbutton pin
int buttonState = 0;
void setup(){
  // initialize the pushbutton pin as an input:
  pinMode(buttonPin, INPUT);
  Serial.begin(9600);
}
void loop(){
  // read the state of the pushbutton value:
  buttonState = digitalRead(buttonPin);
if (buttonState == HIGH) {
    float temp = (analogRead(A0) * resolution) * 100; // converting the value obtained into temperature
    Serial.print("TEMPRATURE = ");
    Serial.print(temp);
    Serial.print("*C");
    Serial.println(" ");
    delay (500);
  }
}
```

Compile and upload the code above to the Arduino and see the temperature values returned by clicking on the Serial monitor button  . The temperature value is calculated and displayed every time the push button is pushed as shown in Figure below.



3.8 Case study: Getting Room Temperature and Sending the Data to the Cloud

3.8.1 Pre-requirements:

1) Hardware

- LM35 (Temperature sensor)
- USB cable,
- Jumper cables
- Breadboard.
- ESP5266 (NodeMCU)
- Blue and red LED •
- 2 resistors 330 Ohm

2) Software

- Arduino IDE

- ThingSpeak

Cloud storage

ThingSpeak is an IoT analytics platform which enables processing and visualization of IoT data in the cloud. The ThingSpeak platform will be used in subsequent case studies to store and visualise the IoT data.

In this case study, we are going to use the hardware to get the room temperature at intervals of 10 minutes and send those values to the cloud platform, ThingSpeak.

3.8.2 Setting up the hardware (IoT testing without cloud)

You will need the Arduino Uno, the USB A to B cable to connect the Arduino to your PC/Laptop Temperature sensor LM35, Jumper cables and a breadboard.

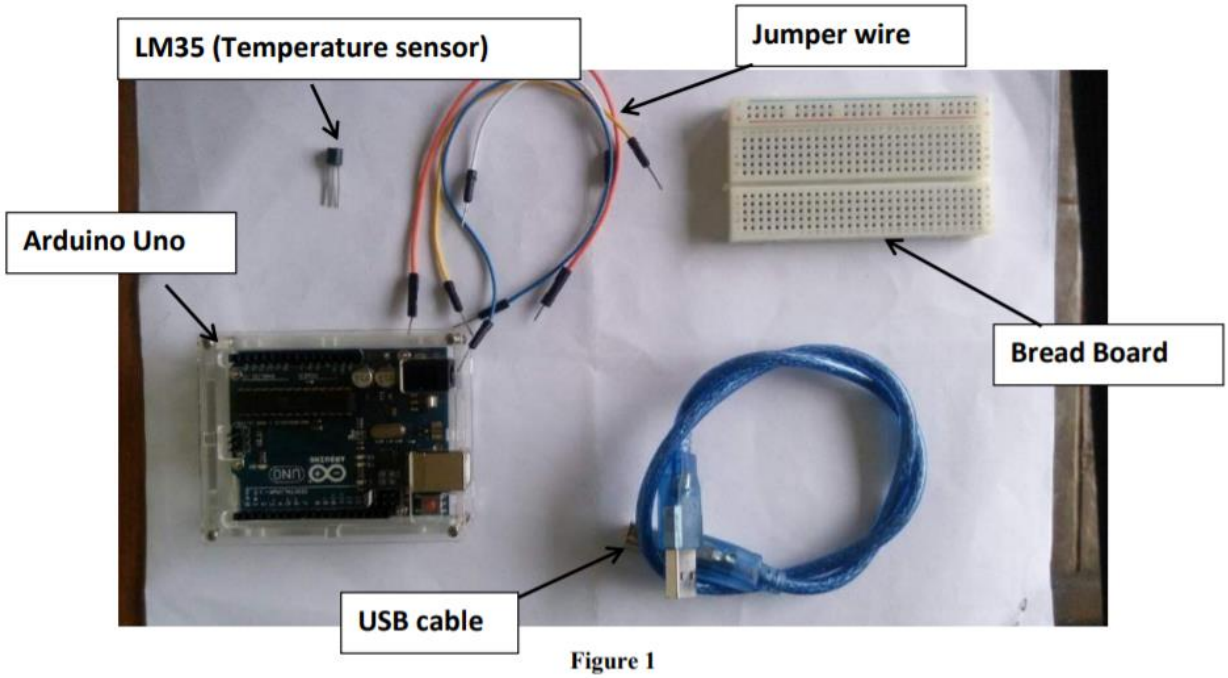


Figure 1

Temperature sensor factsheet:

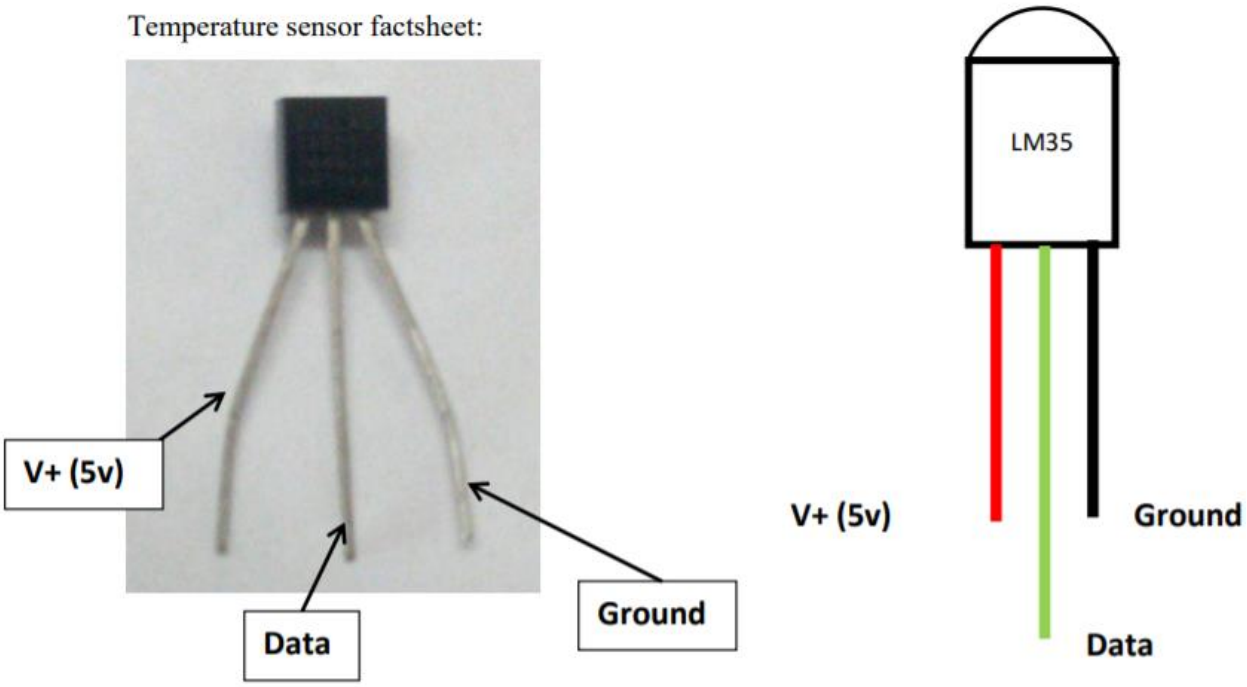
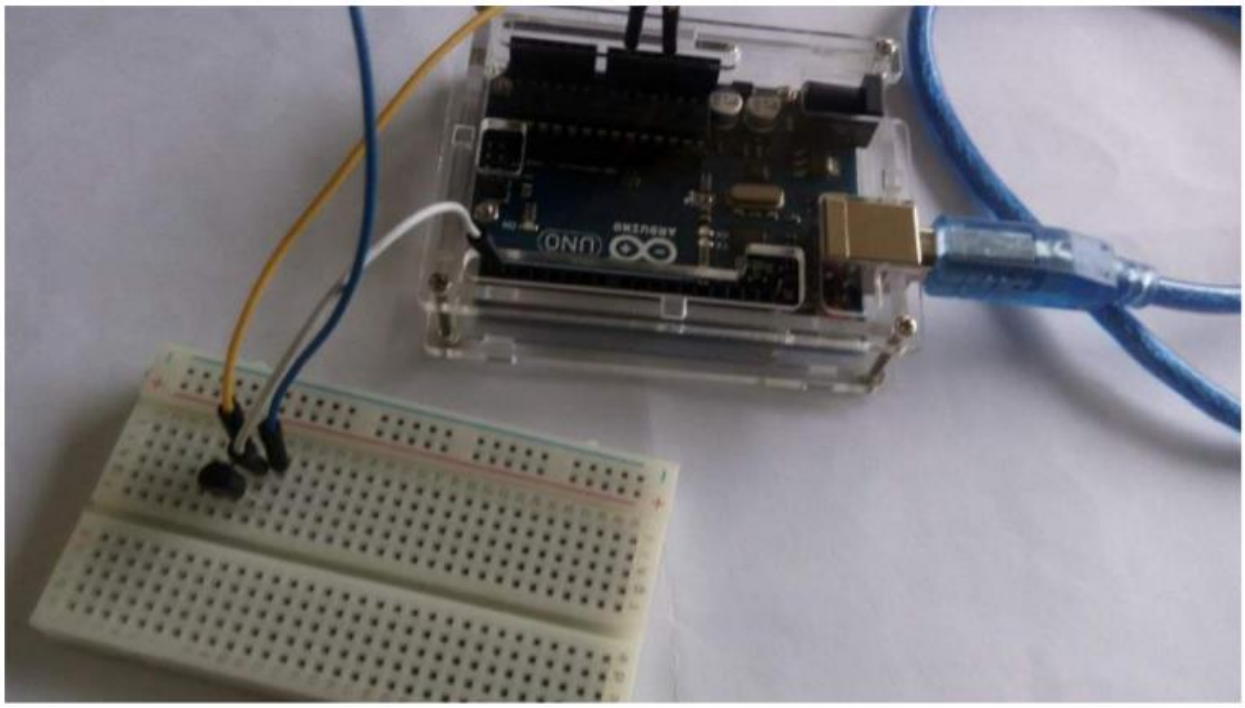


Figure 2

Facing the flat part of the sensor, the pins are labelled as shown in Figure above. The circuit is connected as follows: the positive pin connects to the positive 5v while the last pin connects to the ground pin and the data pin connects to the analog pin A0 of the Arduino. The sensor will be linked to the Arduino by jumper wires which are connected to the breadboard as shown below.



Next step is to write the code to get the temperature values from the sensor. Note that the sensor only sends electrical signal to the Arduino. The sketch will have to interpret these values and return the equivalent in degrees Celsius. For that purpose, we are going to use a formula to convert the electrical values to temperature values.

This can be summarized according to the pseudo code below.

```
BEGIN
INITIALISE
REPEAT
        Read value from pin A0
        Apply formula to convert readings to degrees Celsius
        Send data to Serial port
        Wait 1 second to send next temperature value
END-REPEAT
END
```

The sketch resulting from the above pseudocode is shown below:

```
float resolution=3.3/1023;// 3.3 is the supply voltage  & 1023 is max analog read value
void setup() {
Serial.begin(9600);
}
void loop(){
 float temp = (analogRead(A0) * resolution) * 100; // converting the value obtained into temperature
Serial.print("TEMPRATURE = ");
Serial.print(temp);
Serial.print("*C");
Serial.println();
delay(1000);
}
```

Code Listing 1

Compile and upload the code to the Arduino and see the temperature values returned by clicking on the Serial monitor button . This will load the serial monitor and display the temperature values every second as shown in figure below.

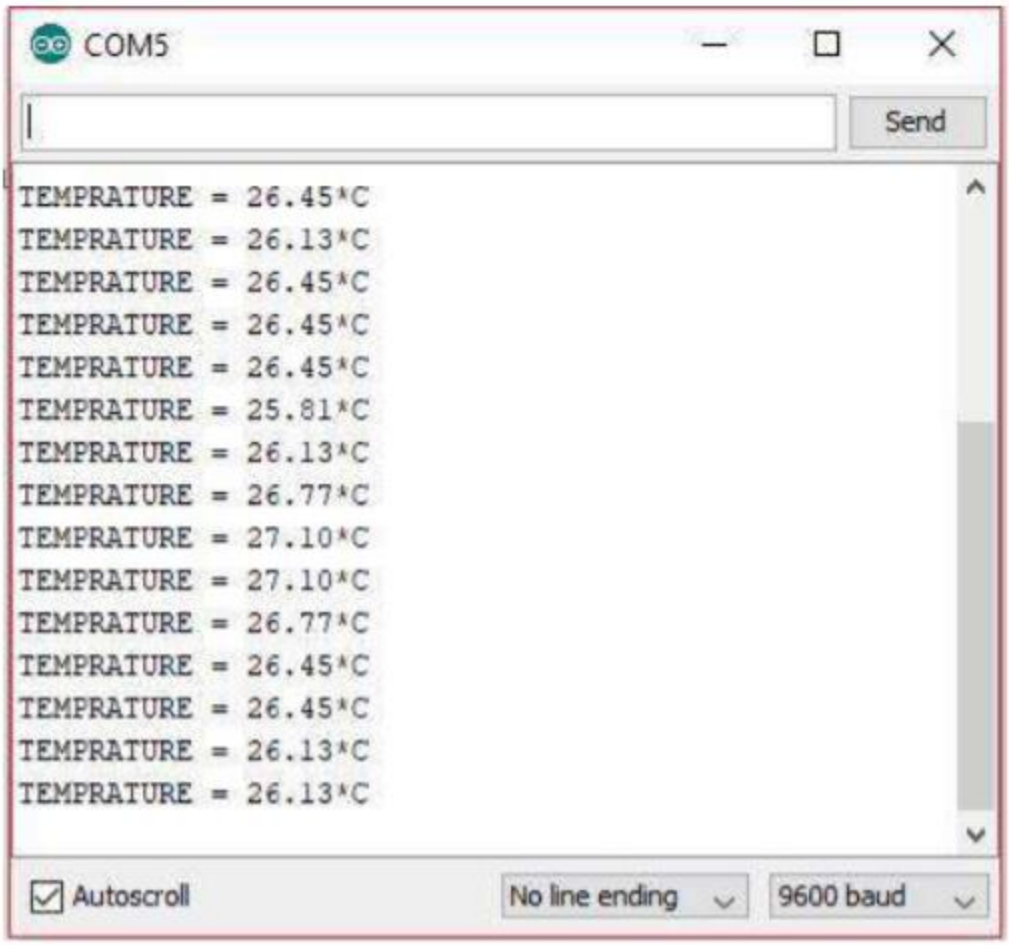


Figure 4

3.8.3 Sending the Data to the Cloud

Go to ThingSpeak (https://thingspeak.com/) website and Sign up for a new account. Click on the 'Get Started for Free' button.

Figure 5

This will load the Sign up page. Fill in your details and click on continue. You will receive an activation email. Follow instructions to activate your account. When you are prompted to activate Matlab license, click on 'No Thanks' button.

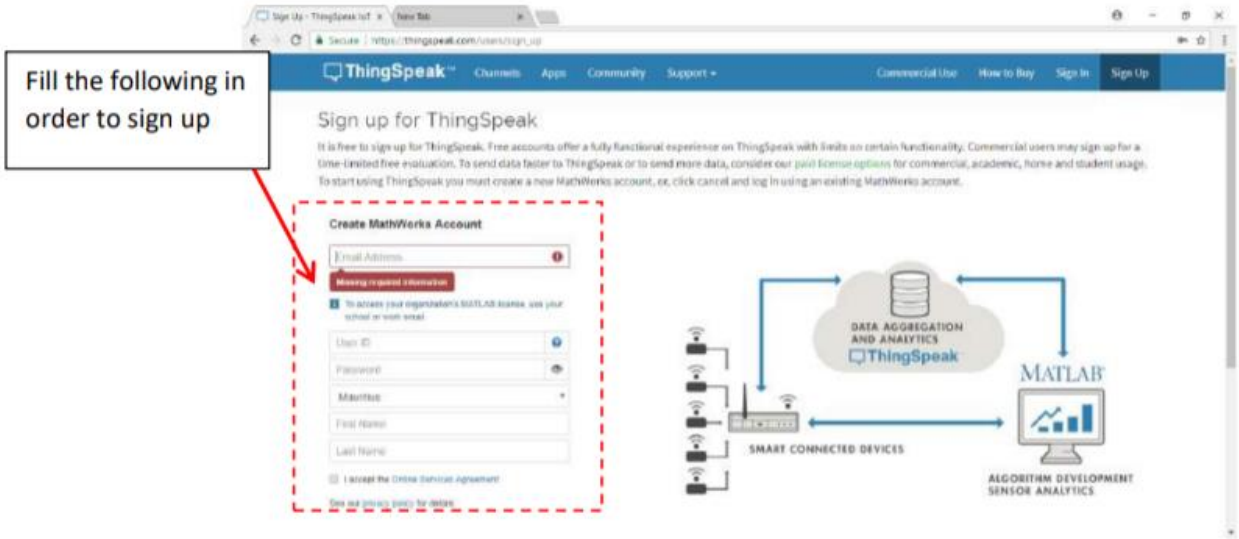


Figure 6

Go to the ThingSpeak site again and log in. Next we need to add a new channel in order to retrieve the data from the NodeMCU and post it online.

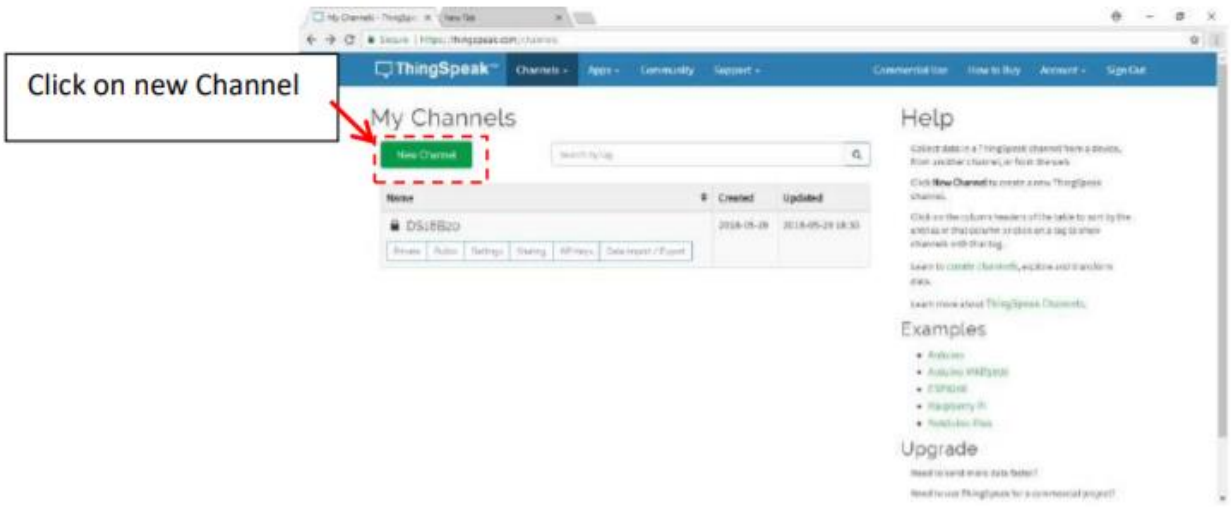


Figure 7

Fill Name as LM35 and Field 1 as Temperature as shown below:

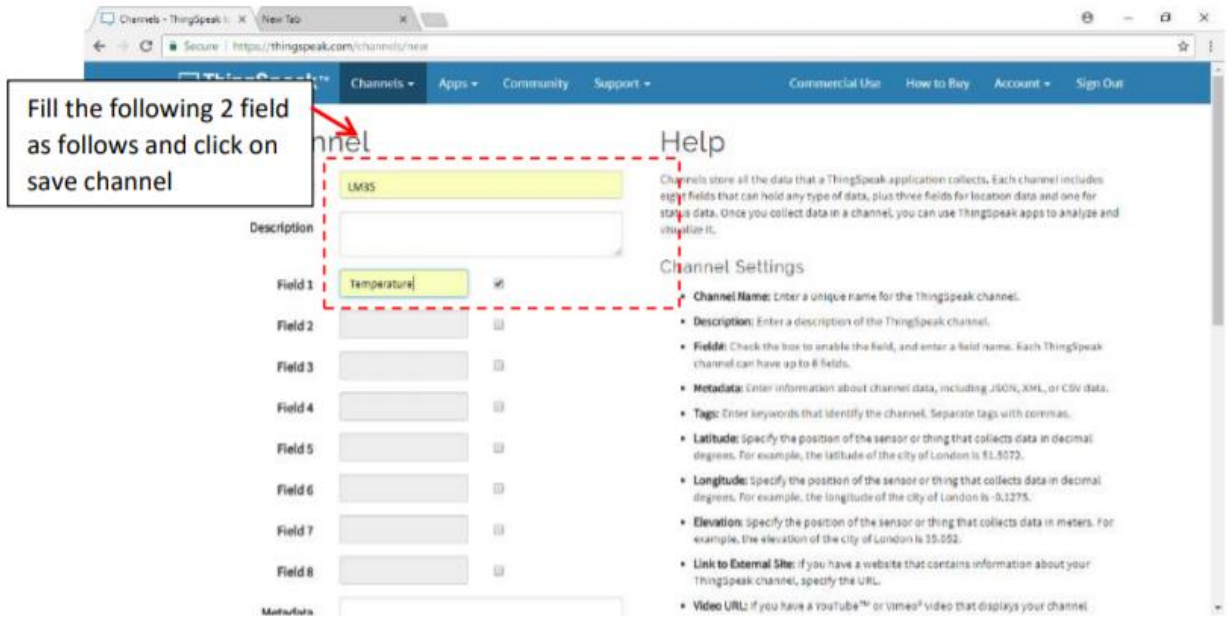


Figure 8

Once this is done, click on API key and copy and paste the Write and Read API key in Code Listing 2 below.

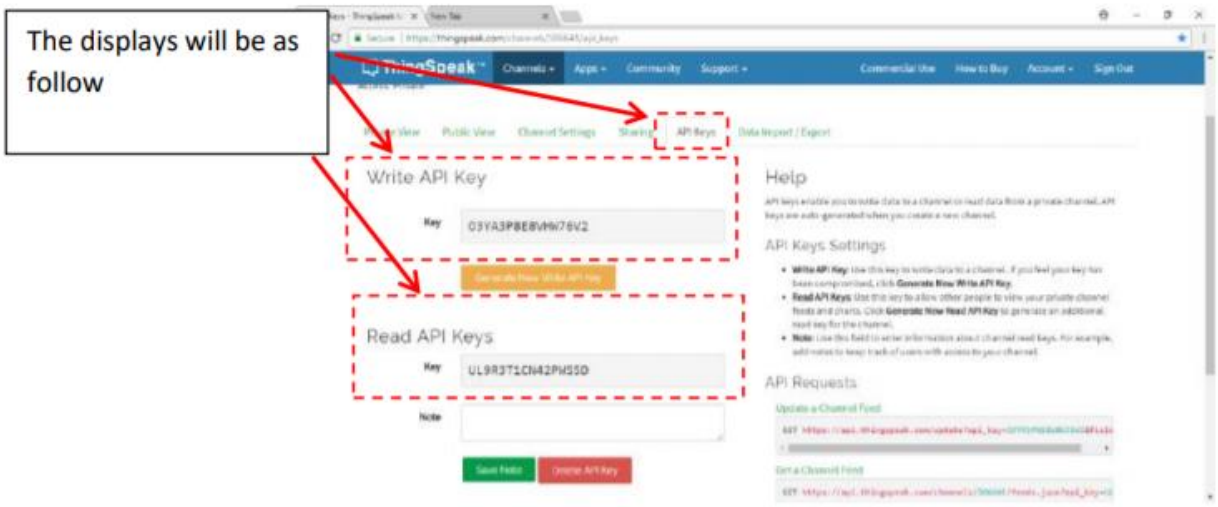

The displays will be as follow
ThingSpeak
Channels
Apps
Community
Support
Commercial Use
How to Buy
Account
Sign Out
Private View
Public View
Channel Settings
Sharing
API Keys
Data Import / Export
Write API Key
Key
03YA3P8E8VM676V2
Read API Keys
Key
UL9R3T1CN42PW55D
Note
Save Note
Delete API Key
Help
API Keys Settings
API Requests

Figure 9

```
#include <ESP8266WiFi.h>
String apiKey ="your API key here"; // the API key as show above in the diagram
const char* MY_SSID = "your SSID here"; // your WIFI name
const char* MY_PWD = "your SSID password here";// your WIFI password


const char* server = "api.thingspeak.com";
float resolution=3.3/1023;// 3.3 is the supply volt  & 1023 is max analog read value
int RedLED = 2;               //outpu pin for LED
int BlueLED = 4;              //outpu pin for LED
WiFiClient client;
 void setup() {
  pinMode(BlueLED,OUTPUT);
  pinMode(RedLED,OUTPUT);


  Serial.begin(115200);
  WiFi.disconnect();
  delay(10);
  WiFi.begin(MY_SSID, MY_PWD);
```

```
 Serial.println();
 Serial.println();
 Serial.print("Connecting to ");
 Serial.println(MY_SSID);

 WiFi.begin(MY_SSID, MY_PWD);

 while (WiFi.status() != WL_CONNECTED) {
  delay(500);
  Serial.print(".");
 }
 Serial.println("");
 Serial.print("NodeMcu connected to wifi...");
 Serial.println(MY_SSID);
 Serial.println();
}
 void loop() {
 float temp = (analogRead(A0) * resolution) * 100; // converting the value obtained into temperature

 if (client.connect(server,80)) // NodeMCU use Port 80 to send data
 {
  String tsData = apiKey;
       tsData +="&field1=";
       tsData += String(temp);
       tsData += "\r\n\r\n";

   client.print("POST /update HTTP/1.1\n");
   client.print("Host: api.thingspeak.com\n");
   client.print("Connection: close\n");
   client.print("X-THINGSPEAKAPIKEY: "+apiKey+"\n");
```

```
   client.print("Content-Type: application/x-www-form-urlencoded\n");
   client.print("Content-Length: ");
   client.print(tsData.length());
   client.print("\n\n");
   client.print(tsData);

   Serial.print("Temperature: ");
   Serial.print(temp);
   Serial.println("°C");
   Serial.println("uploaded to Thingspeak server....");
   if (temp <30)
   {
    digitalWrite(RedLED, LOW);   // turn the LED on (HIGH is the voltage level)
    digitalWrite(BlueLED, HIGH);    // turn the LED off by making the voltage LOW
   }
   else if (temp > 30)
   {
    digitalWrite(RedLED, HIGH);   // turn the LED on (HIGH is the voltage level)
    digitalWrite(BlueLED, LOW);    // turn the LED off by making the voltage LOW
   }
 }
 client.stop();

 Serial.println("Waiting to upload next reading...");
 Serial.println();
 delay(15000); // thingspeak needs minimum 15 sec delay between updates
}
```

Code Listing 2

Three parameters need to be changed:

```
String apiKey ="your API key here"; // the API key as show above in the diagram
const char* MY_SSID = "your SSID here"; // your WIFI name
const char* MY_PWD = "your SSID password here";// your WIFI password
```

Crash Course on NodeMCU

In order to communicate with the cloud service ThingSpeak, we will have to learn how to use the ESP8266 board (NodeMcu). The NodeMcu is an open source hardware that allows us to send data over the internet via a WIFI chip. This hardware can be programmed using the Arduino IDE, but we will have to install some drivers first.

Note: Make sure the latest version of Arduino is installed on your machine.

Step 1: Connect NodeMCU to the Computer Use the USB cable to connect the NodeMCU to the computer. The blue onboard LED will flicker once when powered up.

Step 2: Install the COM/Serial Port Driver

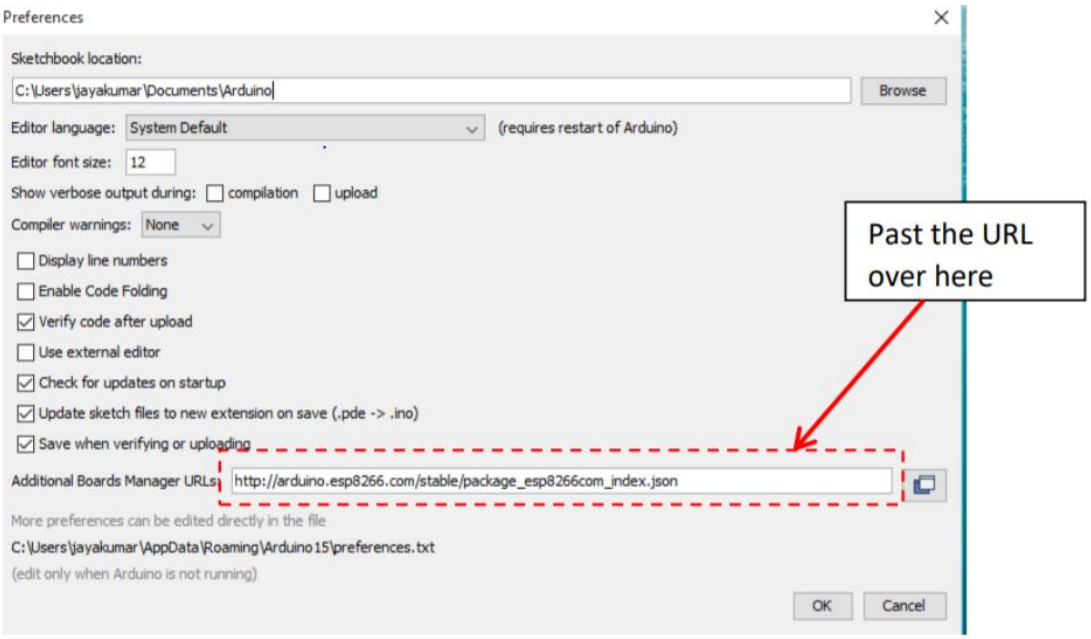
In order to upload code to the ESP8266 and use the serial console, connect a micro-usb data cable to ESP8266 IoT Board and your PC/Laptop. Download and install the relevant driver from the links provided below.

- The new version NodeMCUv1.0 comes with the CP2102 serial chip. Download and install the driver from: https://www.silabs.com/products/development-tools/....
- The NodeMCUv0.9 comes with the CH340 serial chip. Download and install the driver from: https://github.com/nodemcu/nodemcu-devkit/tree/mas...

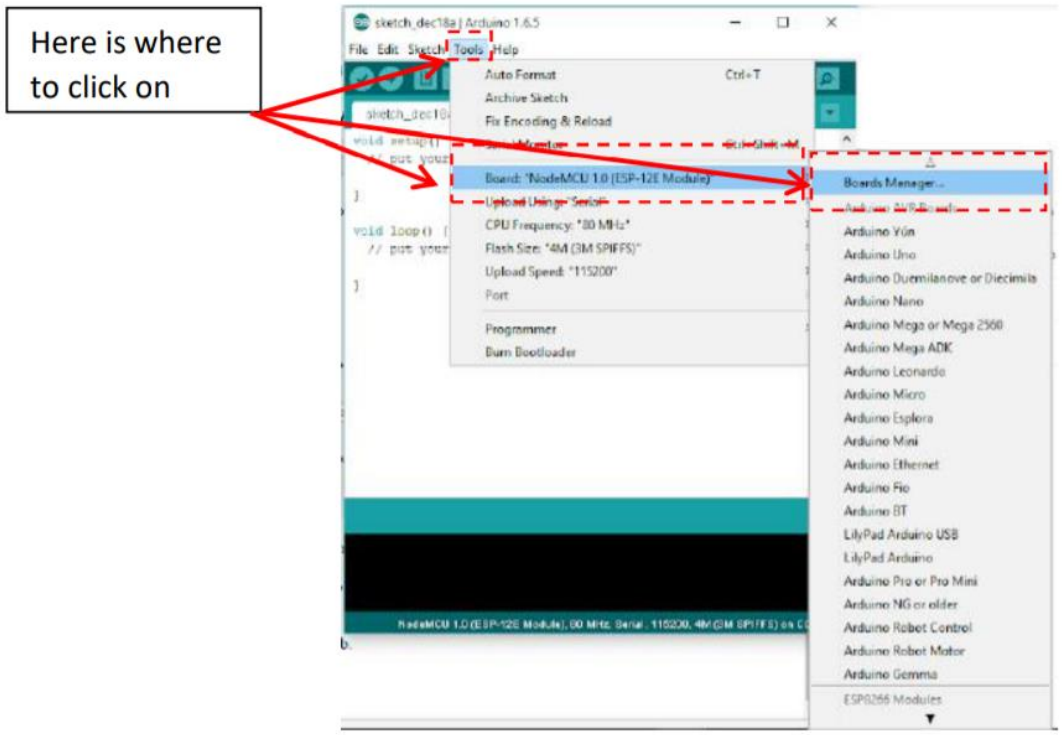
Step 3: Set up NodeMCU in Arduino IDE (requires v1.6.4 or greater)

- Open the Arduino IDE
- Go to **files** and click on the **preference** in the Arduino IDE:
- Write the url http://arduino.esp8266.com/stable/package_esp8266com_index.json

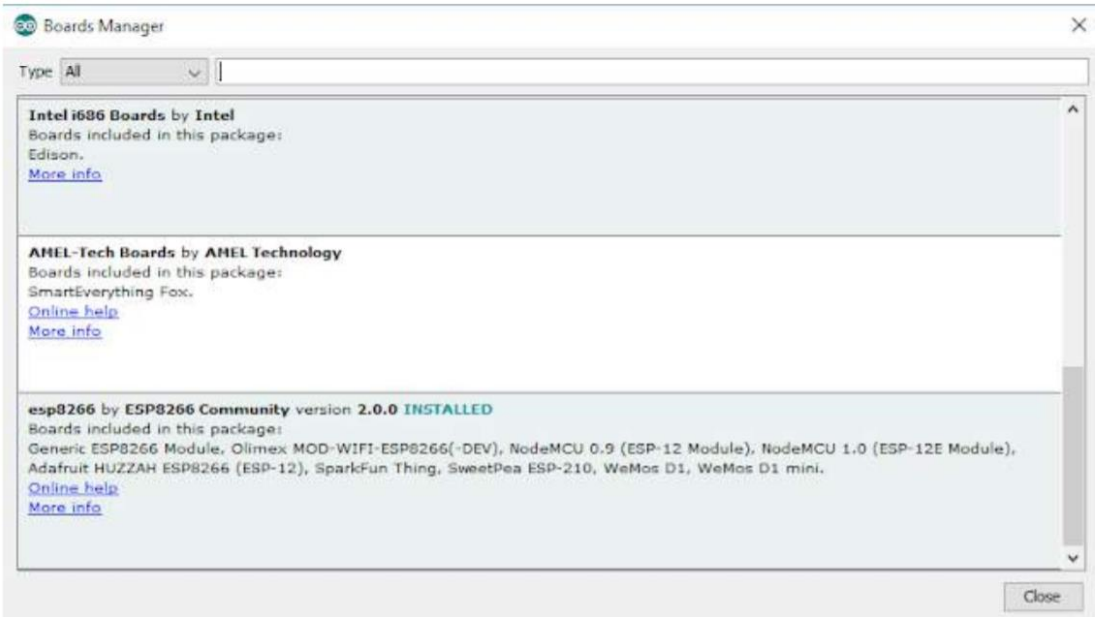
the in the Additional boards Manager URLS as shown below.



- click OK to close the preference Tab.
- After completing the above steps, go to **Tools** and **board**, and then select **board Manager**

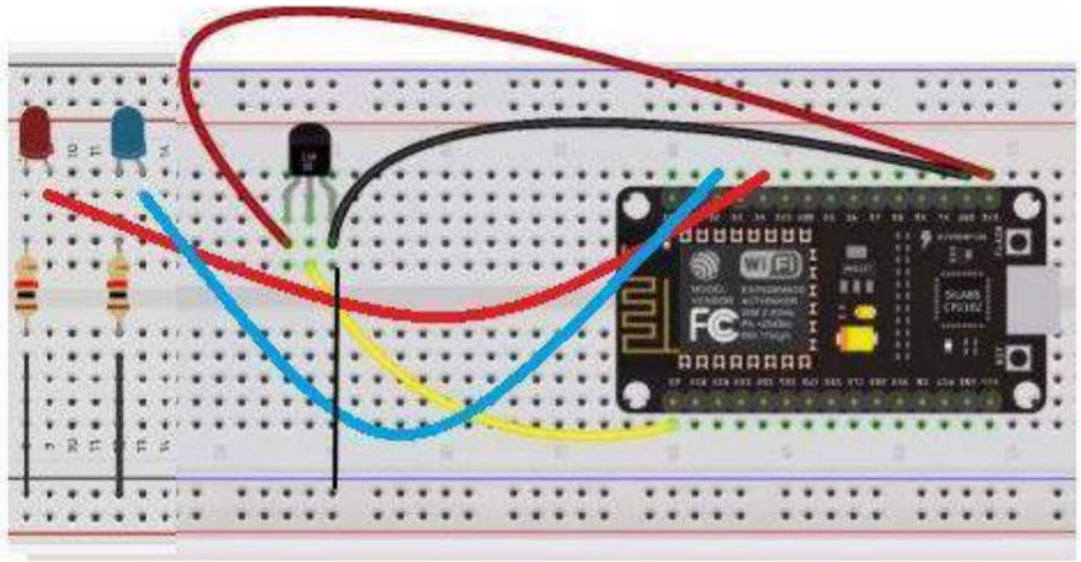


- Navigate to esp8266 by esp8266 community and install the software for Arduino.
- Once all the above process been completed everything is set to program our NodeMcu (esp8266) with Arduino IDE.

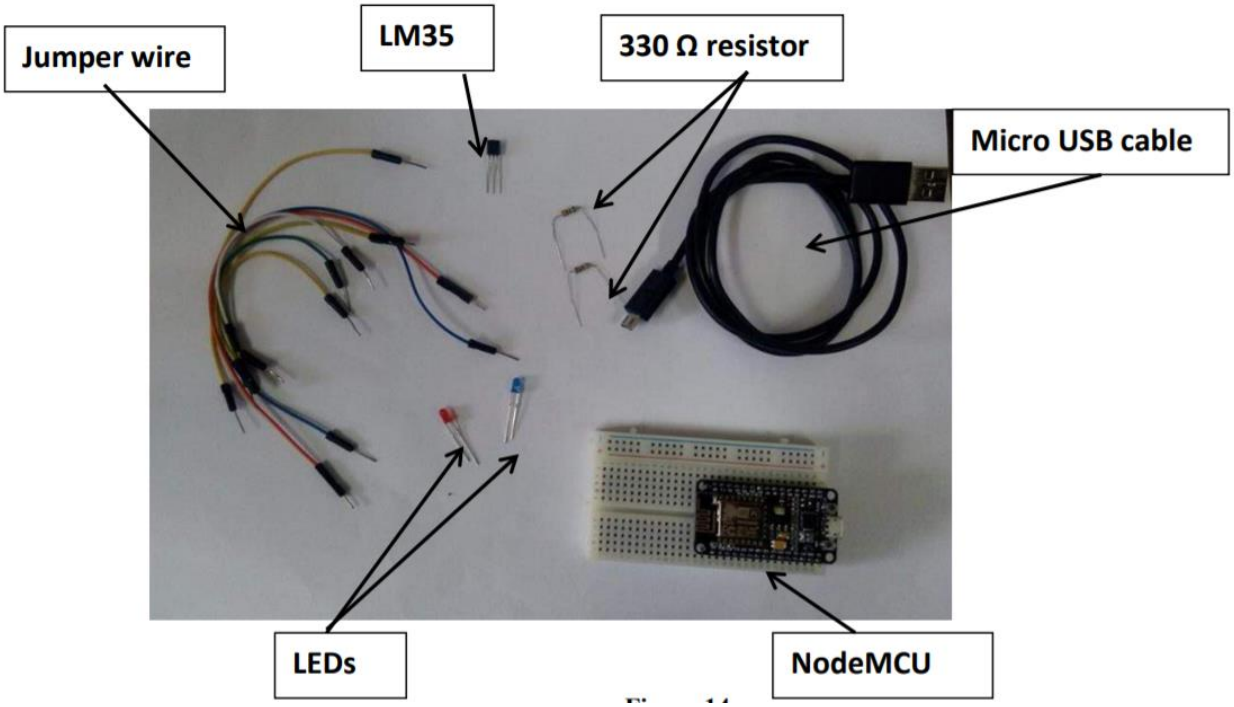


Sending Temperature to ThingSpeak

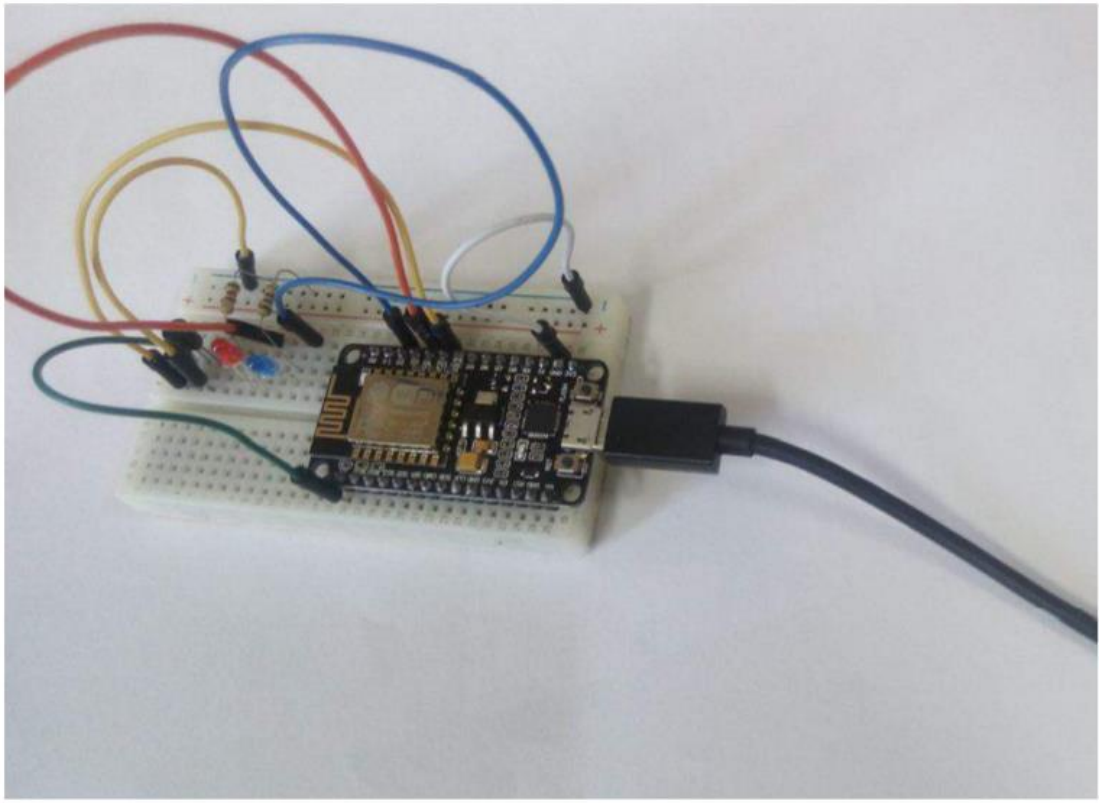
Circuit connection:



The different components needed:



The circuit is connected as follow:



Connect the above circuit to your PC/Laptop and upload the code in code listing 2 via the Arduino IDE to the NodeMcu. Open ThingSpeak dashboard to see the following graph being generated.

CHAPTER 4: CLOUD COMPUTING FUNDAMENTALS

4.1Introduction

Cloud computing is a group of IT services which have been presented to a person on the network with a leased basis current ability to scale up or down their service requirements. Usually Cloud Computing services are delivered by way of an alternative party provider - the master of the infrastructure.

IBM defines cloud computing as being simply “the cloud,” is the delivery of ondemand computing resources—everything from applications to data centres—online using a payfor-use basis (IBM, 2018). Webopedia's defines cloud computing as being usually typically a kind of computing that utilizes sharing computing resources as opposed to having local servers or personal devices to handle applications (Webopedia, 2018).

In cloud computing, the word cloud (also phrased as "the cloud") is used as a metaphor for "the web," therefore the phrase cloud computing means "a form of Internet-based computing," where different services — such as servers, storage and applications — are shipped to an organization's computers and devices over the Internet.

4.2 CLOUD COMPUTING ARCHITECTURE

There are basically 3 layers in cloud computing. Companies use it differently based on their needs. The 3 layers are application, platform and lastly infrastructure. These layers are usually presented in the form of a pyramid with infrastructure at the bottom; platform in the middle; and application at the top of the pyramid.

4.2.1 The Bottom Layer

The bottom layer which is infrastructure is also known as ‘**infrastructure as a service’** (IaaS). This is where the things start and where people begin to build. This is the layer where the cloud hosting lives. The examples of company that provides Cloud infrastructure are Amazon Web Services, GoGrid, and the Rackspace Cloud. Cloud infrastructure is also known to work as a deliver computer infrastructure. Most companies, in this part will operate their own

infrastructure. It will allow them to give more services and features and also give more control than other layers in Cloud Pyramid.

4.2.2 The Middle Layer

The middle layer which is platform is also known as '**platform as a service'** (PaaS). The examples of company and product of Cloud Platform are Google App Engine, Heroku, Mosso (now the Rackspace CloudSites offering), Engine Yard, Joyent or force.com. In contrast to the Cloud Application, this layer is where the users build up the increase of flexibility and control. There are strengths and weakness of characteristic in this Cloud Platform. The strength of Cloud Platform is that it has more control than cloud Application and it also good for developers with a certain position target. Meanwhile the weakness is that sometimes it depends more on Cloud Infrastructure Providers and sometimes it also sticks to the platform ability only. For example, building an app, involves downloading and installing Android SDK (4GB) and other related libraries like maven, Google APIs and so on. Instead, users can use an instance from an already setup platform to start building the app. However, users will be limited to libraries and version of SDK preinstalled.

4.2.3The Top Layer

The top layer which is application is also known as '**software as a service'** (SaaS). In this layer, the users are really limited to what the application can do. The part of company that is involved is the public email providers such as Gmail, Hotmail, Yahoo Mail, etc. Most companies use services in this particular Cloud layer. Usually, users can only get the pre-defined functions and cannot access more than that. Therefore, users can use the application as it appears; and they have no knowledge or any control to the application. The advantages however are that it is free, easy to use and it offers a lot of different services including Google Docs, Sheet, Presentation, CRM, SalesForce and so on

4.3 CLOUD SERVICE MODELS

In line with the different types of services offered, cloud computing can be considered to incorporate three layers: **software** being a service (SAAS), **platform** being a Service (PAAS), and **infrastructure** like a Service (IAAS) (Iyer & Henderson, 2010; Han, 2010, Mell & Grance,

2010). Infrastructure as a Service (IaaS) is the lowest layer that delivers basic infrastructure support service. The middle layer, Platform as a Service (PaaS), provides environment for hosting end-user's applications. Software like a Service (SaaS) may be the topmost layer which comes with a complete application offered as service at will.

4.3.1Software as a Service (SaaS)

Software as a Service (SaaS) (Lin, 2012) is described as software which is deployed over the internet. With SaaS, a provider licenses a credit application to customers either to be a service at the moment, by having a subscription, in the-pay-as-you-go model, or (increasingly) at no cost if there is opportunity to generate revenue from different streams, like from advertisement or user list sales.

Applications could possibly be offered through Internet as browser applications or they are often downloaded and synchronized with user devices. SaaS offers compelling benefits. It simplifies licensing. In fact, the consumer does not need to get a software license in the least. This is the task on the provider. There is also no requirement to calculate maximum capacity. It outsources the tiresome task of application, preservation and upgrades and ties consumer costs to handling, which lowers flat charge and principal investment.

This rapid growth suggests that SaaS has decided to become a commonplace within every organization and therefore it is significant to be aware of what SaaS is, where it works and security issues. The core technology of SaaS is dedicated to its multi-tenant architecture. Chong and Carraro (2006) characterized SaaS as –Software deployed being a hosted service and accessed online. So as to provide proficient and successful services to SaaS consumers, the SaaS providers have got to design their application architecture as –multi-tenant, scalable, efficient, and configurable (Chong & Carraro, 2006).

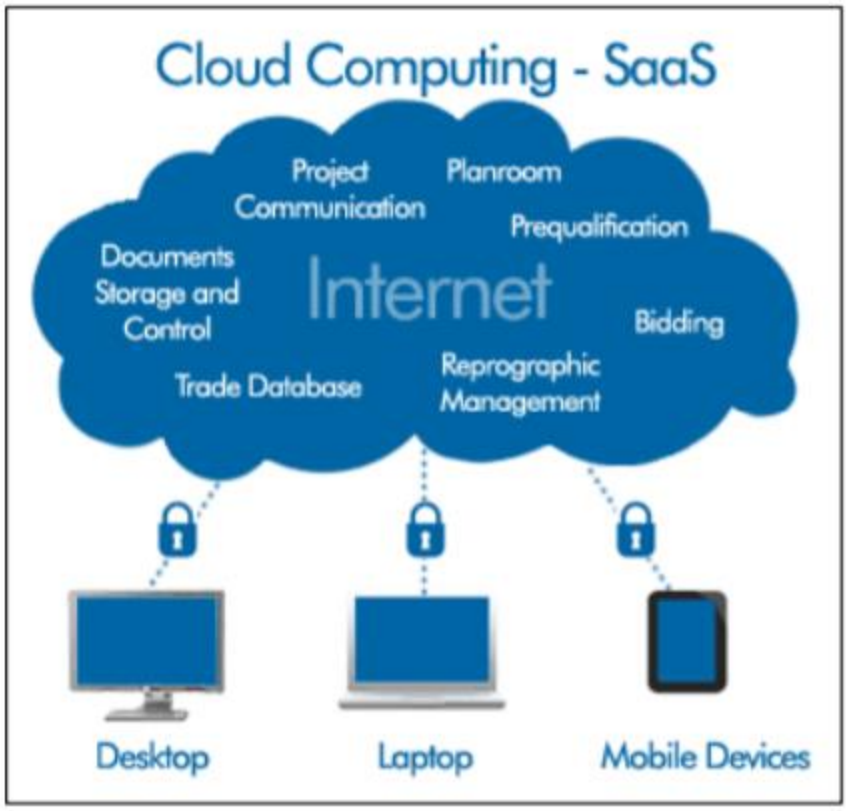


Figure 4 – SaaS (Mirza, 2016)

4.3.2 Platform as a Service (PaaS)

Platform as a service (PaaS) (Beri, 2015) may be the delivery of a computing platform and solution stack like a service. It facilitates deployment of applications without worrying about cost and complexity of purchasing and managing the underlying hardware and software layers, that would be providing every facilities required to offer the complete life cycle to create and deliver web applications and services entirely available from the web itself. Cloud platforms become run-time environments which support a couple of programming languages. They offer additional services such as reusable components and libraries that you can get as objects and application programming interfaces.

Figure 5 – PaaS (Dhiman, 2017)

4.3.3 Infrastructure as a Service (IaaS)

Infrastructure as a Service (IaaS) (Lin, 2012) can be a method of delivering Cloud Computing infrastructure – servers, storage, network and operating systems – being an on-demand service. Rather than purchasing servers, software, data center space or network equipment, clients instead buy those resources being a fully outsourced service at will. Cloud consumers directly use IT infrastructures provided from the IaaS cloud.

Virtualization is expansively used in IaaS cloud so that users can integrate/decompose physical property within an ad-hoc manner to meet increasing or shrinking resource demand from cloud customers. It is surely an evolution of VPS offerings and merely supplies a mechanism to consider advantage of hardware along with other physical resources without capital investment or physical administrative requirements. The benefits of services at this stage are that there are not many restrictions for the consumer. There might be challenges including dedicated hardware but any software program can run within the IaaS context.

4.4 VIRTUALISATION AND RESOURCE MANAGEMENT

Virtualization is the process of simulating hardware, so that several operating systems could be run on a single machine. Most servers are used only about 10 – 20 % if dedicated to a single client. In order to maximize the usage of servers in the cloud, VMs (a VM - Virtual Machine – is the simulation of hardware to create an environment similar to that of physical hardware. Popular

examples include VMWare, VirtualBox) are used. One server could be hosting several VMs as shown in Figure below and each VM is dedicated to one client only. VMs could then be migrated to others servers based on resource management (CPU, Memory or Storage usage) or Locality (Content Delivery Network). The main aim of virtualization is to provide strong isolation, security, performance and simplicity.

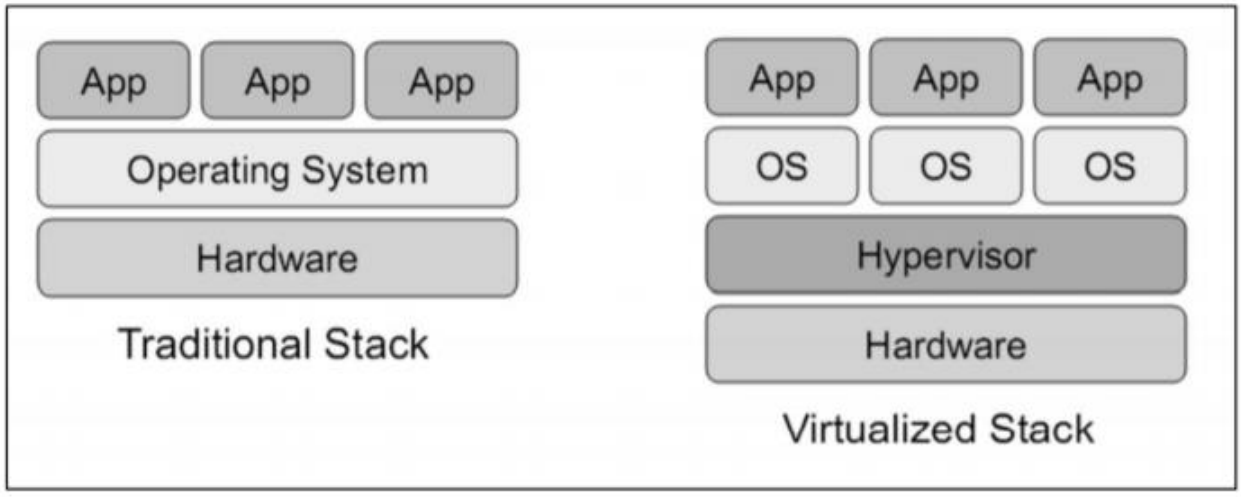


Figure 6 – Traditional VS Virtualised Stack

4.5Types of Virtualization Technology

4.5.1 Hardware Partition

In this approach, hardware (CPU, Memory and storage) is partitioned by using a Partition Controller. Each partition has their own CPU/Memory and independent Operating System installed. However, the disadvantage of this method is the lack of flexibility for the management of the resources in real-time. Once the system is set-up, and a user finds that there is need, for example, of more memory, the system cannot be reconfigured.

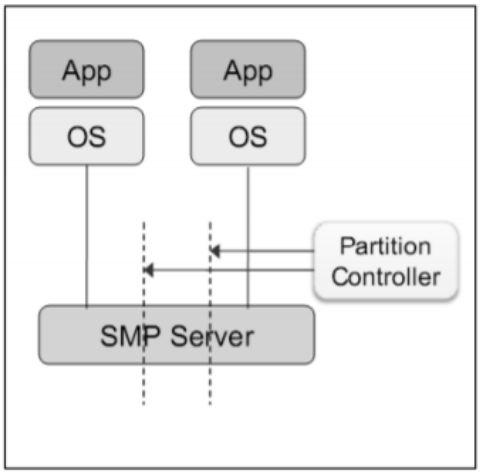


Figure 7 – Hardware Partitioning

4.5.2 Virtual Machine Monitoring (VMM)

In this approach, an application known as VMM is installed on the Host OS, and which allows to create several VM having each their own OS. In this type of virtualization, the kernel of host OS is not modified, as compared to Hardware partition. Another advantage of this method is that it allows each VM to have an OS independent of the host OS as shown in Figure below. However, the downsides of this type is the low efficiency and high cost of hardware instruction translation.

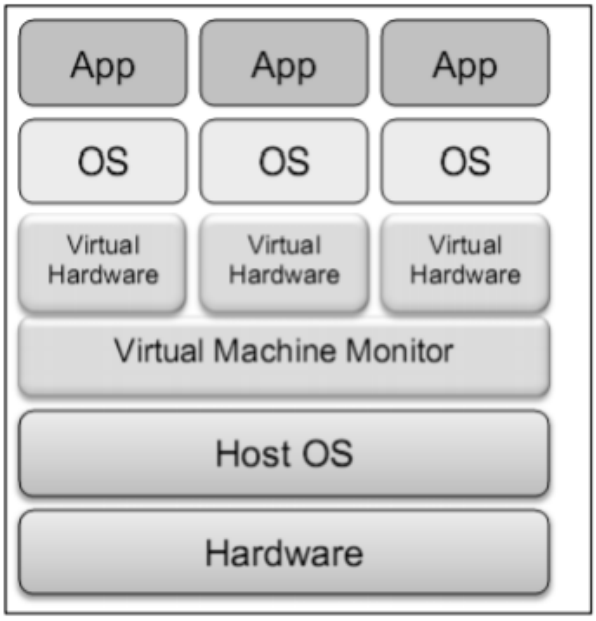


Figure 8 – Virtual Machine Monitor

4.5.3 OS Virtualization

In this approach, one OS instance is installed on a single host. A virtualization platform is installed on top of the host OS. The virtualization platform then offers the possibility to create containers to host virtual OS as shown in Figure below. The benefits of this deployment is the low cost and possibility of running hundreds of VPS (Virtual Private Server) on a single server.

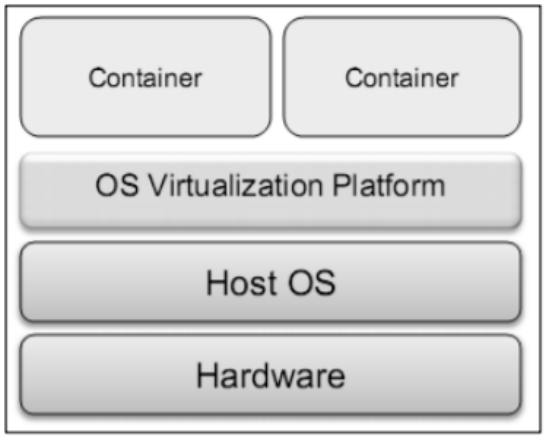


Figure 10 – OS Virtualisation

4.6 Typical IoT Cloud Platforms

1. Thingworx 8 IoT Platform:

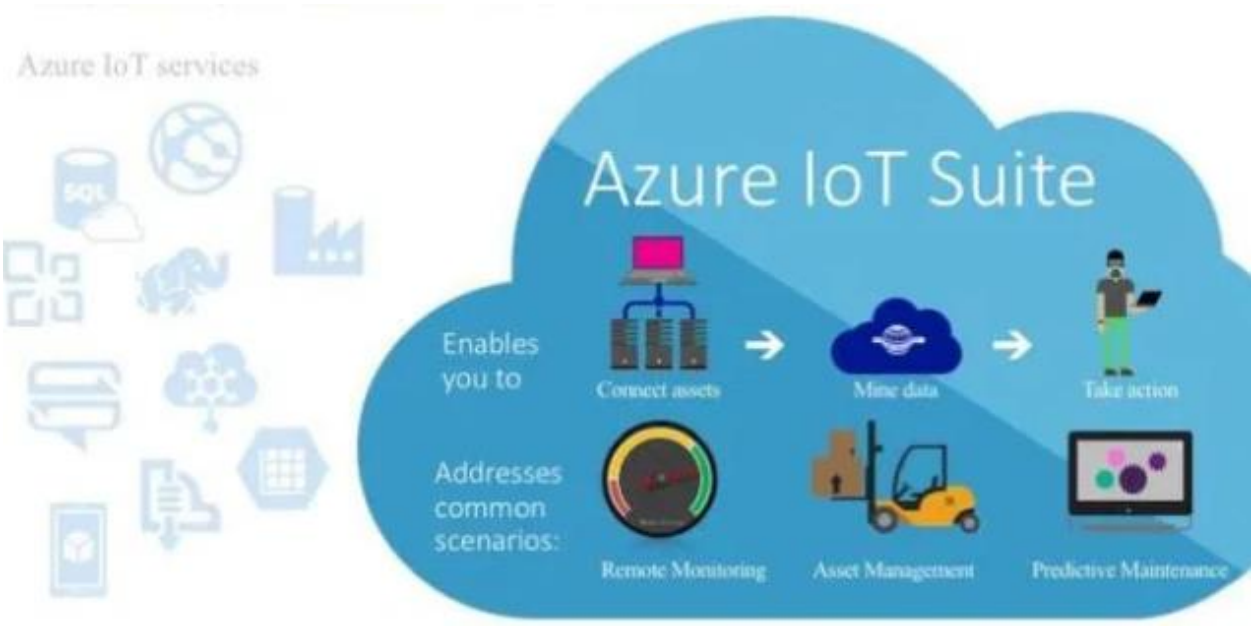
Thingworx is one of the leading IoT platforms for industrial companies, which provides easy connectivity for devices. Thingworx 8 is a better, faster, easier platform, providing the functionality to build, deploy, and extend industrial projects and apps.

It offers basic features, such as:

- Easy connectivity with electronic devices, like sensors and RFIDs
- You can work remotely once you are done with the setup
- Pre-built widgets for the dashboard
- Remove Complexity of the project
- Integrated machine learning

2. Microsoft Azure IoT Suite

Microsoft Azure provides multiple services to create IoT solutions. It enhances your profitability and productivity with pre-built connected solutions.



Azure IoT Suite provides features like:

- Easy Device Registry.
- Dashboards and visualization
- Real-time streaming

3. Google Cloud's IoT Platform

Google has an end-to-end platform for Internet-of-Things solutions. It allows you to easily connect, store, and manage IoT data. This platform helps you to scale your business.

Google Cloud's IoT platform provides the following features:

- Provides huge storage
- Cuts cost for server maintenance
- Business through a fully protected, intelligent, and responsive IoT data

- Efficient and scalable
- Analyze big data

4. IBM Watson IoT Platform

IBM Watson is a powerful platform backed by IBM 's the Bluemix and hybrid cloud PaaS (platform as a service) development platform. By providing easy sample apps and interfaces for IoT services, they make it accessible to beginners.

Users can get the following features:

- Real-time data exchange
- Secure Communication
- Recently added data sensor and weather data service

5. AWS IoT Platform

Amazon made it much easier for developers to collect data from sensors and Internet-connected devices. They help you collect and send data to the cloud and analyze that information to provide the ability to manage devices.

Main features of the AWS IoT platform are:

- Device management
- Secure gateway for devices
- Authentication and encryption

6. Cisco IoT Cloud Connect

Cisco Internet of Things accelerates digital transformation and actions from your data. Cisco IoT Cloud Connect is a mobile, cloud-based suite. It offers solutions for mobile operators to provide phenomenal IoT experience.

7. Salesforce IoT Cloud

Salesforce IoT Cloud is powered by Salesforce Thunder. It gathers data from devices, websites, applications, and partners to trigger actions for real-time responses

Key features of Salesforce IoT Cloud:

- Enhanced data collection
- Real-time event processing
- Technology optimization

8. Kaa IoT Platform

Kaa is an open-source, multipurpose, middleware platform for complete end-to-end IoT development and smart devices. It reduces cost, risk, and market time. Also, Kaa offers a range of IoT tools that can be easily plugged in and implemented in IoT use cases.

It provides many features that make it unique, such as:

- Reduce development time
- Open source and free
- Easy and direct device implementation
- Reduce marketing time
- Handle millions of devices

9. Oracle IoT Platform

Oracle offers real-time Internet of Things data analysis, endpoint management, and high speed messaging where the user can get real-time notification directly on their devices.

Features offering to users

- Secure and scalable
- Real-time insight
- Integrated
- Faster to market

10. Thingspeak IoT Platform

Thingspeak is an open-source platform that allows you to collect and store sensor data to the cloud. It provides you the app to analyze and visualize your data in Matlab. You can use Arduino, Raspberry Pi, and Beaglebone to send sensor data. You can create a separate channel to store data.

UNIT 5: INTRODUCTION TO PYTHON AND RASPBERRY PI PROGRAMMING

5.1 Python Programming

5.1.1 INTRODUCTION

Python is a powerful general-purpose programming language. It is used in web development, data science, creating software prototypes, and so on. In this tutorial, you will learn to install and run Python on your computer. Once we do that, we will also write our first Python program.

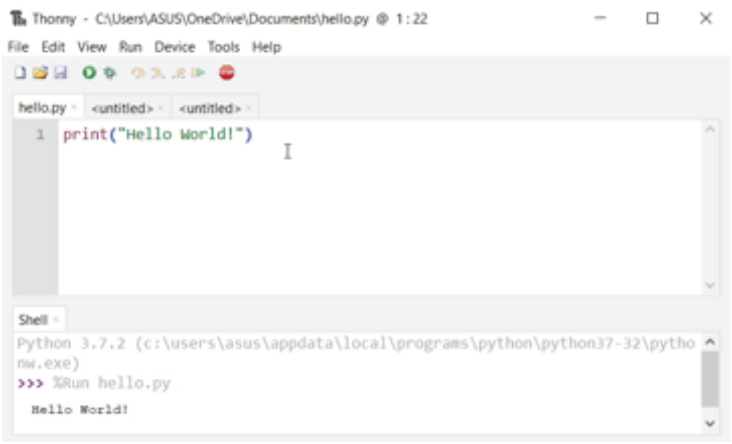
Python is a cross-platform programming language, which means that it can run on multiple platforms like Windows, macOS, Linux, and has even been ported to the Java and .NET virtual machines. It is free and open-source. Even though most of today's Linux and Mac have Python pre-installed in it, the version might be out-of-date. So, it is always a good idea to install the most current version.

- **Install Python IDE**

The easiest way to run Python is by using Thonny IDE. The Thonny IDE comes with the latest version of Python bundled in it. So you don't have to install Python separately. Follow the following steps to run Python on your computer.

Download Thonny IDE. Run the installer to install Thonny on your computer. Go to: File > New. Then save the file with .py extension. You can give any name to the file. However, the file name should end with .py

Write Python code in the file and save it.



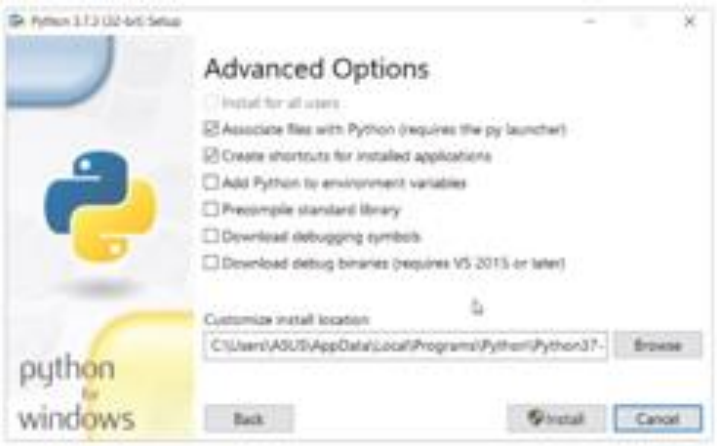
Then Go to Run > Run current script or simply click F5 to run it.

----------Install Python Separately

If you don't want to use Thonny, here's how you can install and run Python on your computer.

Download the latest version of Python. Run the installer file and follow the steps to install Python. During the install process, check Add Python to environment variables. This will add Python to environment variables, and you can run Python from any part of the computer.

Also, you can choose the path where Python is installed.



Once you finish the installation process, you can run Python

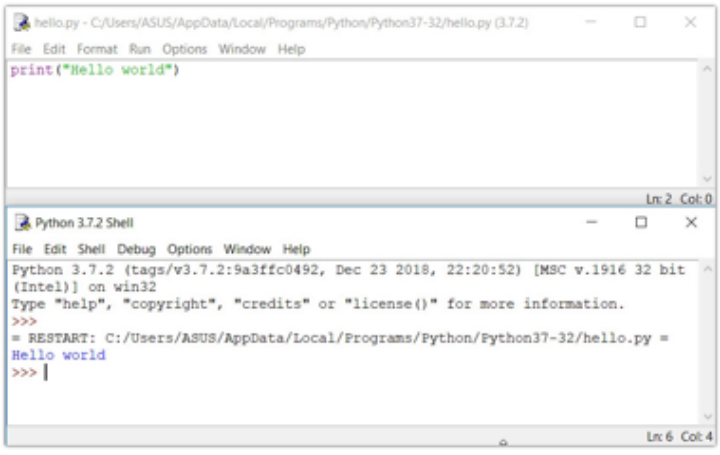
Once Python is installed, typing `python` in the command line will invoke the interpreter in immediate mode. We can directly type in Python code, and press Enter to get the output. Try typing in `1 + 1` and press enter. We get `2` as the output. This prompt can be used as a calculator. To exit this mode, type `quit()` and press enter.

```
Microsoft Windows [Version 10.0.17134.648]
(c) 2018 Microsoft Corporation. All rights reserved.

C:\Users\ASUS>python
Python 3.7.2 (tags/v3.7.2:9a3ffc0492, Dec 23 2018, 22:20:52
) [MSC v.1916 32 bit (Intel)] on win32
Type "help", "copyright", "credits" or "license" for more i
nformation.
>>> 1 + 1
2
>>> quit()

C:\Users\ASUS>
```

----------Run Python in the Integrated Development Environment (IDE)

We can use any text editing software to write a Python script file. We just need to save it with the .py extension. But using an IDE can make our life a lot easier. By the way, when you install Python, an IDE named IDLE is also installed. You can use it to run Python on your computer. It's a decent IDE for beginners. When you open IDLE, an interactive Python Shell is opened.

Now you can create a new file and save it with .py extension. For example, hello.py. Write Python code in the file and save it. To run the file, go to Run > Run Module or simply click F5.



i

- **Your first Python Program**

Let's create a very simple program called `Hello World`. A "Hello, World!" is a simple program that outputs `Hello, World!` on the screen.

```
print("Hello, world!")
```

Then, run the file. You will get the following output.

```
Hello, world!
```

Congratulations! You just wrote your first program in Python

- **Python Keywords and Identifiers**

In this tutorial, you will learn about keywords (reserved words in Python) and identifiers (names given to variables, functions, etc.). We cannot use a keyword as a **variable** name, **function** name or any other identifier. They are used to define the syntax and structure of the Python language. These words have their own specific meaning in python programming. In Python, keywords are case sensitive. There are 33 keywords in Python 3.7. This number can vary slightly over the course of time. All the keywords except `True`, `False` and `None` are in lowercase.

- **List of all the keywords:**

Keywords in Python programming language				
False	await	else	import	pass
None	break	except	in	raise
True	class	finally	is	return
and	continue	for	lambda	try

as	def	from	nonlocal	while
assert	del	global	not	with
async	elif	if	or	yield

- **Python Statement, Indentation and Comments**

In this tutorial, you will learn about Python statements, why indentation is important and use of comments in programming.

- **Python Statement**

Instructions that a Python interpreter can execute are called statements. For example, `a = 1` is an assignment statement. `if` statement, `for` statement, `while` statement, etc. are other kinds of statements which will be discussed later.

- **Multi-line statement**

In Python, the end of a statement is marked by a newline character. But we can make a statement extend over multiple lines with the line continuation character (\). For example:

```
a = 1 + 2 + 3 + \
    4 + 5 + 6 + \
    7 + 8 + 9
```

This is an explicit line continuation. In Python, line continuation is implied inside parentheses `( )`, brackets `[ ]`, and braces `{ }`. For instance, we can implement the above multi-line statement as:

```
a = (1 + 2 + 3 +
    4 + 5 + 6 +
```

```
    7 + 8 + 9)
```

Here, the surrounding parentheses *( )* do the line continuation implicitly. Same is the case with *[ ]* and *{ }*. For example:

```
colors = ['red',
          'blue',
          'green']
```

We can also put multiple statements in a single line using semicolons, as follows:

```
a = 1; b = 2; c = 3
```

- **Python Indentation**

Most of the programming languages like C, C++, and Java use braces *{ }* to define a block of code. Python, however, uses indentation.

A code block (body of a function, loop, etc.) starts with indentation and ends with the first unindented line. The amount of indentation is up to you, but it must be consistent throughout that block.

Generally, four whitespaces are used for indentation and are preferred over tabs. Here is an example.

```
for i in range(1,11):
    print(i)
    if i == 5:
        break
```

The enforcement of indentation in Python makes the code look neat and clean. This results in Python programs that look similar and consistent.

Indentation can be ignored in line continuation, but it's always a good idea to indent. It makes the code more readable. For example:

```
if True:
   print('Hello')
   a = 5
```

and

```
if True: print('Hello'); a = 5
```

both are valid and do the same thing, but the former style is clearer.

Incorrect indentation will result in IndentationError.

- **Python Comments**

Comments are very important while writing a program. They describe what is going on inside a program, so that a person looking at the source code does not have a hard time figuring it out.

In Python, we use the hash (#) symbol to start writing a comment. It extends up to the newline character. Comments are for programmers to better understand a program. Python Interpreter ignores comments.

```
#This is a comment
```

```
 # print out Hello print('Hello')
```

- **Multi-line comments**

We can have comments that extend up to multiple lines. One way is to use the hash(#) symbol at the beginning of each line. For example:

```
#This is a long comment #and it extends #to multiple lines
```

Another way of doing this is to use triple quotes, either `'''` or `"""`. These triple quotes are generally used for multi-line strings. But they can be used as a multi-line comment as well. Unless they are not docstrings, they do not generate any extra code.

```
"""This is also a

perfect example of

multi-line comments"""
```

- **Docstrings in Python**

A docstring is short for documentation string. Python docstrings (documentation strings) are the string literals that appear right after the definition of a function, method, class, or module.

Triple quotes are used while writing docstrings. For example:

```
def double(num):
    """Function to double the value"""
    return 2*num
```

Docstrings appear right after the definition of a function, class, or a module. This separates docstrings from multiline comments using triple quotes. The docstrings are associated with the object as their `__doc__` attribute. So, we can access the docstrings of the above function with the following lines of code:

```
def double(num):
    """Function to double the value"""
    return 2*num
print(double.__doc__)
```

Output

Function to double the value

- **Python Variables, Constants and Literals**

In this tutorial, you will learn about Python variables, constants, literals and their use cases.

---------**Python Variables**

A variable is a named location used to store data in the memory. It is helpful to think of variables as a container that holds data that can be changed later in the program. For example,

number = 10

Here, we have created a variable named *number*. We have assigned the value 10 to the variable.

You can think of variables as a bag to store books in it and that book can be replaced at any time.

number = 10

number = 1.1

Initially, the value of *number* was 10. Later, it was changed to 1.1.

Note: In Python, we don't actually assign values to the variables. Instead, Python gives the reference of the object(value) to the variable.

Assigning values to Variables in Python

As you can see from the above example, you can use the assignment operator = to assign a value to a variable. Example 1: Declaring and assigning value to a variable

website = "apple.com"

print(website)

Output

apple.com

In the above program, we assigned a value apple.com to the variable *website*. Then, we printed out the value assigned to *website* i.e. apple.com

Assigning multiple values to multiple variables

```
a, b, c = 5, 3.2, "Hello"

print (a) print (b) print (c)
```

If we want to assign the same value to multiple variables at once, we can do this as:

```
x = y = z = "same"

print (x) print (y) print (z)
```

The second program assigns the same string to all the three variables *x*, *y* and *z*.

---------Constants

A constant is a type of variable whose value cannot be changed. It is helpful to think of constants as containers that hold information which cannot be changed later.

You can think of constants as a bag to store some books which cannot be replaced once placed inside the bag.

Assigning value to constant in Python

In Python, constants are usually declared and assigned in a module. Here, the module is a new file containing variables, functions, etc which is imported to the main file. Inside the module, constants are written in all capital letters and underscores separating the words.

Example 3: Declaring and assigning value to a constant

Create a constant.py:

```
PI = 3.14

GRAVITY = 9.8
```

Create a main.py:

```
import constant

print(constant.PI)

print(constant.GRAVITY)
```

Output

```
3.14

9.8
```

In the above program, we create a constant.py module file. Then, we assign the constant value to *PI* and *GRAVITY*. After that, we create a main.py file and import the constant module. Finally, we print the constant value.

Note: In reality, we don't use constants in Python. Naming them in all capital letters is a convention to separate them from variables, however, it does not actually prevent reassignment.

- **String literals**

A string literal is a sequence of characters surrounded by quotes. We can use both single, double, or triple quotes for a string. And, a character literal is a single character surrounded by single or double quotes.

Example 7: How to use string literals in Python?

```
strings = "This is Python"

char = "C"
```

```
multiline_str = """This is a multiline string with more than one line code."""

unicode = u"\u00dcnic\u00f6de"

raw_str = r"raw \n string"

print(strings)  print(char)  print(multiline_str)  print(unicode)  print(raw_str)
```

----------Boolean literals

A Boolean literal can have any of the two values: `True` or `False`.

Example 8: How to use boolean literals in Python?

```
x = (1 == True)

y = (1 == False)

a = True + 4

b = False + 10

print("x is", x)

print("y is", y)

print("a:", a)

print("b:", b)
```

Output

```
x is True

y is False

a: 5

b: 10
```

- **Python Data Types**

Every value in Python has a datatype. Since everything is an object in Python programming, data types are actually classes and variables are instance (object) of these classes. There are various data types in Python. Some of the important types are listed below.

---------Python Numbers

Integers, floating point numbers and complex numbers fall under Python numbers category. They are defined as int, float and complex classes in Python. We can use the type() function to know which class a variable or a value belongs to. Similarly, the isinstance() function is used to check if an object belongs to a particular class.

```
a = 5

print(a, "is of type", type(a))

a = 2.0

print(a, "is of type", type(a))

a = 1+2j

print(a, "is complex number?", isinstance(1+2j,complex))
```

- **Python List**

List is an ordered sequence of items. It is one of the most used data type in Python and is very flexible. All the items in a list do not need to be of the same type. Declaring a list is pretty straight forward. Items separated by commas are enclosed within brackets [ ].

```
a = [1, 2.2, 'python']
```

We can use the slicing operator [ ] to extract an item or a range of items from a list. The index starts from 0 in Python.

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

# a[2] = 15print("a[2] = ", a[2])

# a[0:3] = [5, 10, 15]print("a[0:3] = ", a[0:3])

# a[5:] = [30, 35, 40]print("a[5:] = ", a[5:])
```

Lists are mutable, meaning, the value of elements of a list can be altered.

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

a[2] = 4

print(a)
```

Output

```
[1, 2, 4]
```

- **Python Tuple**

Tuple is an ordered sequence of items same as a list. The only difference is that tuples are immutable. Tuples once created cannot be modified. Tuples are used to write-protect data and are usually faster than lists as they cannot change dynamically. It is defined within parentheses () where items are separated by commas.

```
t = (5,'program', 1+3j)
```

We can use the slicing operator [] to extract items but we cannot change its value.

```
t = (5,'program', 1+3j)

# t[1] = 'program'
```

```
print("t[1] = ", t[1])

# t[0:3] = (5, 'program', (1+3j))

print("t[0:3] = ", t[0:3])

t[0] = 10   # Generates error# Tuples are immutable
```

- **Python Strings**

String is sequence of Unicode characters. We can use single quotes or double quotes to represent strings. Multi-line strings can be denoted using triple quotes, ''' or """.

```
s = "This is a string"

print(s)

s = '''A multiline

string'''

print(s)
```

Just like a list and tuple, the slicing operator [ ] can be used with strings. Strings, however, are immutable.

```
s = 'Hello world!'

# s[4] = 'o' print("s[4] = ", s[4])

# s[6:11] = 'world' print("s[6:11] = ", s[6:11])

# Generates error# Strings are immutable in Python

s[5] ='d'
```

- **Python Set**

Set is an unordered collection of unique items. Set is defined by values separated by comma inside braces { }. Items in a set are not ordered.

```
a = {5,2,3,1,4}

# printing set variable print("a = ", a)

# data type of variable aprint(type(a))
```

Output

```
a =  {1, 2, 3, 4, 5}

<class 'set'>
```

We can perform set operations like union, intersection on two sets. Sets have unique values. They eliminate duplicates.

```
a = {1,2,2,3,3,3}

print(a)
```

Output

```
{1, 2, 3}
```

Since, set are unordered collection, indexing has no meaning. Hence, the slicing operator [] does not work.

```
>>> a = {1,2,3}  >>> a[1]

Traceback (most recent call last):

  File "<string>", line 301, in runcode

  File "<interactive input>", line 1, in <module>
```

```
TypeError: 'set' object does not support indexing
```

- **Python Dictionary**

Dictionary is an unordered collection of key-value pairs. It is generally used when we have a huge amount of data. Dictionaries are optimized for retrieving data. We must know the key to retrieve the value. In Python, dictionaries are defined within braces {} with each item being a pair in the form key:value. Key and value can be of any type.

```
>>> d = {1:'value','key':2}     >>> type(d)

<class 'dict'>
```

We use key to retrieve the respective value. But not the other way around.

```
d = {1:'value','key':2}   print(type(d))

print("d[1] = ", d[1]);

print("d['key'] = ", d['key']);

# Generates error print("d[2] = ", d[2]);
```

- **Conversion between data types**

We can convert between different data types by using different type conversion functions like int(), float(), str(), etc.

```
>>> float(5)

5.0
```

Conversion from float to int will truncate the value (make it closer to zero).

```
>>> int(10.6)
10
>>> int(-10.6)
-10
```

Conversion to and from string must contain compatible values.

```
>>> float('2.5')
2.5
>>> str(25)
'25'
>>> int('1p')
```

We can even convert one sequence to another.

```
>>> set([1,2,3])
{1, 2, 3}
>>> tuple({5,6,7})
(5, 6, 7)
>>> list('hello')
['h', 'e', 'l', 'l', 'o']
```

To convert to dictionary, each element must be a pair:

```
>>> dict([[1,2],[3,4]])
{1: 2, 3: 4}
>>> dict([(3,26),(4,44)])
```

```
{3: 26, 4: 44}
```

- **Python Type Conversion and Type Casting**

  .Type Conversion

The process of converting the value of one data type (integer, string, float, etc.) to another data type is called type conversion. Python has two types of type conversion.

Implicit Type Conversion

Explicit Type Conversion

Implicit Type Conversion

In Implicit type conversion, Python automatically converts one data type to another data type. This process doesn't need any user involvement. Let's see an example where Python promotes the conversion of the lower data type (integer) to the higher data type (float) to avoid data loss.

Example 1: Converting integer to float

```
num_int = 123

num_flo = 1.23

num_new = num_int + num_flo

print("datatype of num_int:",type(num_int))

print("datatype of num_flo:",type(num_flo))

print("Value of num_new:",num_new)
```

Now, let's try adding a string and an integer, and see how Python deals with it.

Example 2: Addition of string(higher) data type and integer(lower) datatype

```
num_int = 123

num_str = "456"

print("Data type of num_int:",type(num_int))

print("Data type of num_str:",type(num_str))

print(num_int+num_str)
```

When we run the above program, the output will be:

In the above program, We add two variables *num_int* and *num_str*. As we can see from the output, we got TypeError. Python is not able to use Implicit Conversion in such conditions.

However, Python has a solution for these types of situations which is known as Explicit Conversion.

Explicit Type Conversion

In Explicit Type Conversion, users convert the data type of an object to required data type. We use the predefined functions like int(), float(), str(), etc to perform explicit type conversion. This type of conversion is also called typecasting because the user casts (changes) the data type of the objects.

Syntax :

```
<required_datatype>(expression)
```

Typecasting can be done by assigning the required data type function to the expression.

Example 3: Addition of string and integer using explicit conversion

```
num_int = 123

num_str = "456"
```

```
print("Data type of num_int:",type(num_int))

print("Data type of num_str before Type Casting:",type(num_str))

num_str = int(num_str)

print("Data type of num_str after Type Casting:",type(num_str))

num_sum = num_int + num_str

print("Sum of num_int and num_str:",num_sum)

print("Data type of the sum:",type(num_sum))
```

In the above program, We add *num_str* and *num_int* variable. We converted *num_str* from string(higher) to integer(lower) type using int() function to perform the addition. After converting *num_str* to an integer value, Python is able to add these two variables. We got the *num_sum* value and data type to be an integer.

- **Python Input, Output and Import**

This tutorial focuses on two built-in functions print() and input() to perform I/O task in Python. Also, you will learn to import modules and use them in your program.

Python provides numerous built-in functions that are readily available to us at the Python prompt. Some of the functions like input() and print() are widely used for standard input and output operations respectively. Let us see the output section first.

---------Python Output Using print() function

We use the print() function to output data to the standard output device (screen). We can also output data to a file, but this will be discussed later. An example of its use is given below.

```
print('This sentence is output to the screen')
```

Another example is given below:

```
a = 5

print('The value of a is', a)
```

Output

```
The value of a is 5
```

In the second print() statement, we can notice that space was added between the string and the value of variable *a*. This is by default, but we can change it.

The actual syntax of the print() function is:

```
print(*objects, sep=' ', end='\n', file=sys.stdout, flush=False)
```

Here, objects is the value(s) to be printed. The sep separator is used between the values. It defaults into a space character. After all values are printed, end is printed. It defaults into a new line.

The file is the object where the values are printed and its default value is sys.stdout (screen). Here is an example to illustrate this.

```
print(1, 2, 3, 4) print(1, 2, 3, 4, sep='*') print(1, 2, 3, 4, sep='#', end='&')
```

Output

```
1 2 3 4

1*2*3*4

1#2#3#4&
```

- **Output formatting**

Sometimes we would like to format our output to make it look attractive. This can be done by using the str.format() method. This method is visible to any string object.

```
>>> x = 5; y = 10

>>> print('The value of x is {} and y is {}'.format(x,y))

The value of x is 5 and y is 10
```

Here, the curly braces {} are used as placeholders. We can specify the order in which they are printed by using numbers (tuple index).

```
print('I love {0} and {1}'.format('bread','butter'))

print('I love {1} and {0}'.format('bread','butter'))
```

Output

```
I love bread and butter

I love butter and bread
```

We can also format strings like the old sprintf() style used in C programming language. We use the % operator to accomplish this.

```
>>> x = 12.3456789

>>> print('The value of x is %3.2f' %x)

The value of x is 12.35

>>> print('The value of x is %3.4f' %x)

The value of x is 12.3457
```

- **Python Input**

Up until now, our programs were static. The value of variables was defined or hard coded into the source code. To allow flexibility, we might want to take the input from the user. In Python, we have the input() function to allow this. The syntax for input() is:

```
input([prompt])
```

where prompt is the string we wish to display on the screen. It is optional.

```
>>> num = input('Enter a number: ')
Enter a number: 10
>>> num
'10'
```

Here, we can see that the entered value 10 is a string, not a number. To convert this into a number we can use int() or float() functions.

```
>>> int('10')
10
>>> float('10')
10.0
```

- **Python Import**

When our program grows bigger, it is a good idea to break it into different modules. A module is a file containing Python definitions and statements. Python modules have a filename and end with the extension .py. We use the import keyword to do this. For example, we can import the math module by typing the following line:

```
import math
```

We can use the module in the following ways:

```
import math

print(math.pi)
```

Output

```
3.141592653589793
```

Now all the definitions inside `math` module are available in our scope. We can also import some specific attributes and functions only, using the `from` keyword. For example:

```
>>> from math import pi

>>> pi3.141592653589793
```

- **Python Operators**

Operators are special symbols in Python that carry out arithmetic or logical computation. The value that the operator operates on is called the operand.

For example:

```
>>> 2+35
```

Here, `+` is the operator that performs addition. `2` and `3` are the operands and `5` is the output of the operation.

----Arithmetic operators

Arithmetic operators are used to perform mathematical operations like addition, subtraction, multiplication, etc.

Operator	Meaning	Example
+	Add two operands or unary plus	x + y+ 2
-	Subtract right operand from the left or unary minus	x - y- 2
*	Multiply two operands	x * y
/	Divide left operand by the right one (always results into float)	x / y
%	Modulus - remainder of the division of left operand by the right	x % y (remainder of x/y)
//	Floor division - division that results into whole number adjusted to the left in the number line	x // y
**	Exponent - left operand raised to the power of right	x**y (x to the power y)

----Comparison operators

Comparison operators are used to compare values. It returns either True or False according to the condition.

Operator	Meaning	Example
>	Greater than - True if left operand is greater than the right	x > y
<	Less than - True if left operand is less than the right	x < y
==	Equal to - True if both operands are equal	x == y
!=	Not equal to - True if operands are not equal	x != y
>=	Greater than or equal to - True if left operand is greater than or equal to the right	x >= y
<=	Less than or equal to - True if left operand is less than or equal to the right	x <= y

---

----Logical operators

Logical operators are the and, or, not operators.

Operator	Meaning	Example
and	True if both the operands are true	x and y

or	True if either of the operands is true	x or y
not	True if operand is false (complements the operand)	not x

Example 3: Logical Operators in Python

```
x = True
y = False
print('x and y is',x and y)
print('x or y is',x or y)
print('not x is',not x)
```

------Bitwise operators

Bitwise operators act on operands as if they were strings of binary digits. They operate bit by bit, hence the name. For example, 2 is 10 in binary and 7 is 111.

In the table below: Let *x* = 10 (0000 1010 in binary) and *y* = 4 (0000 0100 in binary)

Operator	Meaning	Example
&	Bitwise AND	x & y = 0 (0000 0000)
|	Bitwise OR	x | y = 14 (0000 1110)
~	Bitwise NOT	~x = -11 (1111 0101)

^	Bitwise XOR	x ^ y = 14 (0000 1110)
>>	Bitwise right shift	x >> 2 = 2 (0000 0010)
<<	Bitwise left shift	x << 2 = 40 (0010 1000)

-----Assignment operators

Assignment operators are used in Python to assign values to variables. `a = 5` is a simple assignment operator that assigns the value 5 on the right to the variable *a* on the left. There are various compound operators in Python like `a += 5` that adds to the variable and later assigns the same. It is equivalent to `a = a + 5`.

Operator	Example	Equivalent to
=	x = 5	x = 5
+=	x += 5	x = x + 5
-=	x -= 5	x = x - 5
*=	x *= 5	x = x * 5
/=	x /= 5	x = x / 5
%=	x %= 5	x = x % 5

//=	x //= 5	x = x // 5
**=	x **= 5	x = x ** 5
&=	x &= 5	x = x & 5
|=	x |= 5	x = x | 5
^=	x ^= 5	x = x ^ 5
>>=	x >>= 5	x = x >> 5
<<=	x <<= 5	x = x << 5

----Special operators

Python language offers some special types of operators like the identity operator or the membership operator. They are described below with examples.

Identity operators

is and is not are the identity operators in Python. They are used to check if two values (or variables) are located on the same part of the memory. Two variables that are equal does not imply that they are identical.

Operator	Meaning	Example
is	True if the operands are identical (refer to the same object)	x is True

is not	True if the operands are not identical (do not refer to the same object)	x is not True

Example 4: Identity operators in Python

```
x1 = 5

y1 = 5

x2 = 'Hello'

y2 = 'Hello'

x3 = [1,2,3]

y3 = [1,2,3]

# Output: False print(x1 is not y1)

# Output: True print(x2 is y2)

# Output: False print(x3 is y3)
```

Here, we see that *x1* and *y1* are integers of the same values, so they are equal as well as identical. Same is the case with *x2* and *y2* (strings). But *x3* and *y3* are lists. They are equal but not identical. It is because the interpreter locates them separately in memory although they are equal.

--------Membership operators

in and not in are the membership operators in Python. They are used to test whether a value or variable is found in a sequence (string, list, tuple, set and dictionary). In a dictionary we can only test for presence of key, not the value.

Operator	Meaning	Example
in	True if value/variable is found in the sequence	5 in x
not in	True if value/variable is not found in the sequence	5 not in x

Example #5: Membership operators in Python

```
x = 'Hello world'

y = {1:'a',2:'b'}

# Output: True print('H' in x)

# Output: True print('hello' not in x)

# Output: True print(1 in y)

# Output: False print('a' in y)
```

Output

```
True

True

True

False
```

Here, 'H' is in *x* but 'hello' is not present in *x* (remember, Python is case sensitive).

Similarly, 1 is key and 'a' is the value in dictionary *y*. Hence, 'a' in y returns False.

5.1.2 PYTHON FLOW CONTROL

5.1.2.1 Python if...else Statement

Decision making is required when we want to execute a code only if a certain condition is satisfied. The if…elif…else statement is used in Python for decision making.

Python if Statement Syntax

```
if test expression:
    statement(s)
```

Here, the program evaluates the test expression and will execute statement(s) only if the test expression is True. If the test expression is False, the statement(s) is not executed. In Python, the body of the if statement is indicated by the indentation. The body starts with an indentation and the first unindented line marks the end. Python interprets non-zero values as True. None and 0 are interpreted as False.

Python if Statement Flowchart

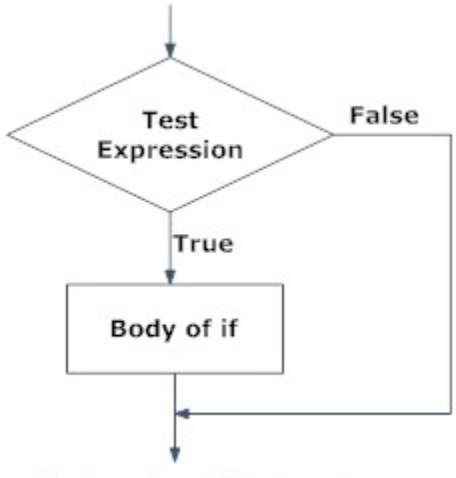


Fig: Operation of if statement

Example: Python if Statement

```
# If the number is positive, we print an appropriate message

num = 3

if num > 0:

   print(num, "is a positive number.")

   print("This is always printed.")

num = -1

if num > 0:

   print(num, "is a positive number.")

   print("This is also always printed.")
```

- **Python if...else Statement**

Syntax of if...else

```
if test expression:

   Body of if

else:

   Body of else
```

The if..else statement evaluates test expression and will execute the body of if only when the test condition is True. If the condition is False, the body of else is executed. Indentation is used to separate the blocks.

- **Python if..else Flowchart**

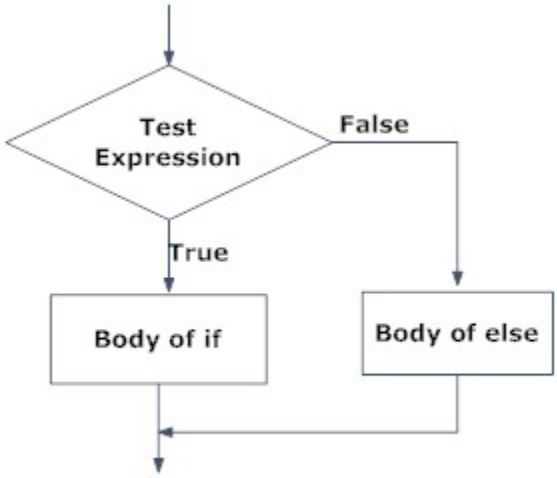


Fig: Operation of if...else statement

Example of if...else

```
# Program checks if the number is positive or negative# And displays an appropriate message

num = 3

# Try these two variations as well. # num = -5# num = 0

if num >= 0:
   print("Positive or Zero")
else:
   print("Negative number")
```

- **Python if...elif...else Statement**

Syntax of if...elif...else

```
if test expression:
    Body of if
elif test expression:
    Body of elif
else:
    Body of else
```

The elif is short for else if. It allows us to check for multiple expressions. If the condition for if is False, it checks the condition of the next elif block and so on. If all the conditions are False, the body of else is executed. Only one block among the several if...elif...else blocks is executed according to the condition. The if block can have only one else block. But it can have multiple elif blocks.

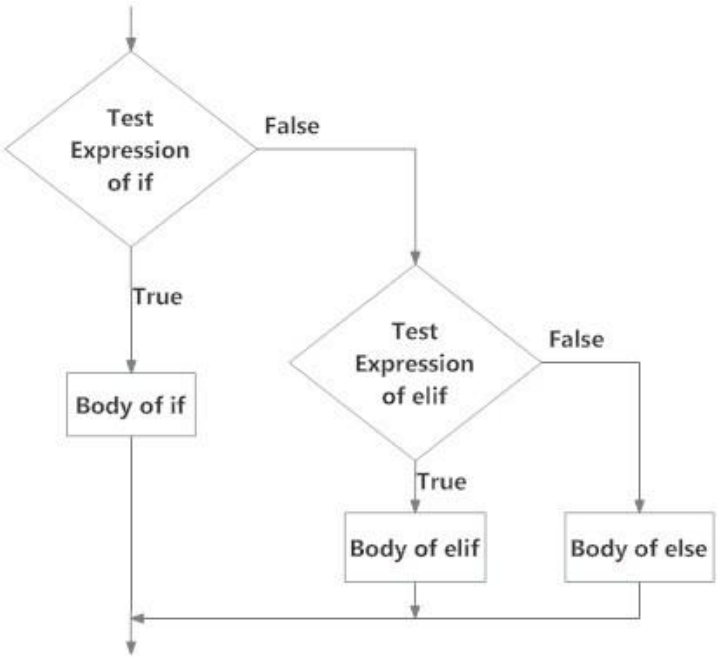


Fig: Operation of if...elif...else statement

Example of if...elif...else

```
'''In this program, we check if the number is positive or negative or zero and

display an appropriate message'''

num = 3.4

# Try these two variations as well:# num = 0# num = -4.5

if num > 0:

    print("Positive number")

elif num == 0:

    print("Zero")

else:

    print("Negative number")
```

When variable *num* is positive, Positive number is printed. If *num* is equal to 0, Zero is printed. If *num* is negative, Negative number is printed.

- **Python Nested if statements**

We can have a if...elif...else statement inside another if...elif...else statement. This is called nesting in computer programming. Any number of these statements can be nested inside one another. Indentation is the only way to figure out the level of nesting. They can get confusing, so they must be avoided unless necessary.

Python Nested if Example

```
'''In this program, we input a number

check if the number is positive or

negative or zero and display
```

```
an appropriate message

This time we use nested if statement'''


num = float(input("Enter a number: "))

if num >= 0:

   if num == 0:

      print("Zero")

   else:

      print("Positive number")else:

   print("Negative number")
```

5.1.2.2 Python for Loop

The for loop in Python is used to iterate over a sequence (list, tuple, string) or other iterable objects. Iterating over a sequence is called traversal.

Syntax of for Loop

```
for val in sequence:

        Body of for
```

Here, val is the variable that takes the value of the item inside the sequence on each iteration. Loop continues until we reach the last item in the sequence. The body of for loop is separated from the rest of the code using indentation.

Flowchart of for Loop

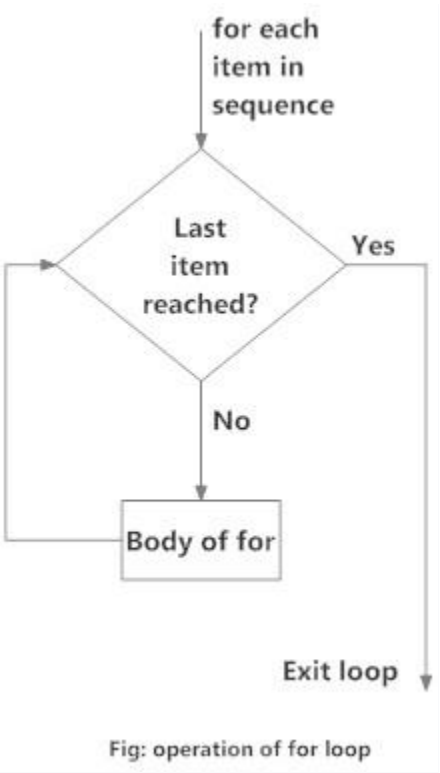


Fig: operation of for loop

Example: Python for Loop

```
# Program to find the sum of all numbers stored in a list

# List of numbers

numbers = [6, 5, 3, 8, 4, 2, 5, 4, 11]

# variable to store the sum

sum = 0

# iterate over the list

for val in numbers:

        sum = sum+val

print("The sum is", sum)
```

The range() function

We can generate a sequence of numbers using range() function. range(10) will generate numbers from 0 to 9 (10 numbers). We can also define the start, stop and step size as range(start, stop,step_size). step_size defaults to 1 if not provided.

The range object is "lazy" in a sense because it doesn't generate every number that it "contains" when we create it. However, it is not an iterator since it supports in, len and __getitem__ operations.

This function does not store all the values in memory; it would be inefficient. So it remembers the start, stop, step size and generates the next number on the go. To force this function to output all the items, we can use the function list(). The following example will clarify this.

```
print(range(10))

print(list(range(10)))

print(list(range(2, 8)))

print(list(range(2, 20, 3)))
```

Output

```
range(0, 10)

[0, 1, 2, 3, 4, 5, 6, 7, 8, 9]

[2, 3, 4, 5, 6, 7]

[2, 5, 8, 11, 14, 17]
```

We can use the range() function in for loops to iterate through a sequence of numbers. It can be combined with the len() function to iterate through a sequence using indexing. Here is an example.

```
# Program to iterate through a list using indexing

genre = ['pop', 'rock', 'jazz']

# iterate over the list using index

for i in range(len(genre)):

        print("I like", genre[i])
```

Output

```
I like pop

I like rock

I like jazz
```

5.1.2.3 Python while Loop

Loops are used in programming to repeat a specific block of code. In this article, you will learn to create a while loop in Python. The while loop in Python is used to iterate over a block of code as long as the test expression (condition) is true. We generally use this loop when we don't know the number of times to iterate beforehand.

Syntax of while Loop in Python

```
while test_expression:

  Body of while
```

In the while loop, test expression is checked first. The body of the loop is entered only if the `test_expression` evaluates to `True`. After one iteration, the test expression is checked again. This process continues until the `test_expression` evaluates to `False`. In Python, the body of the while loop is determined through indentation. The body starts with indentation and the first

unindented line marks the end. Python interprets any non-zero value as True. None and 0 are interpreted as False.

Flowchart of while Loop

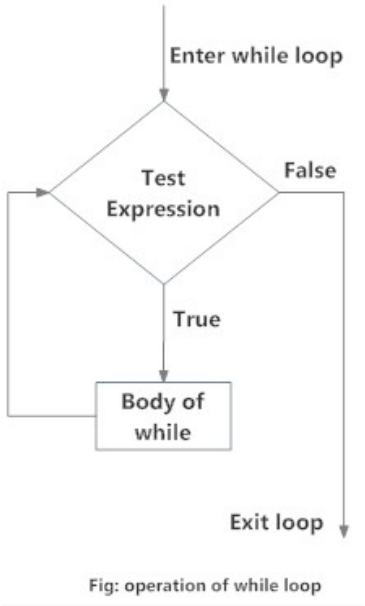


Fig: operation of while loop

Example: Python while Loop

```
# Program to add natural# numbers up to # sum = 1+2+3+...+n

n = 10

# initialize sum and counter

sum = 0

i = 1

while i <= n:

    sum = sum + i

    i = i+1    # update counter

# print the sumprint("The sum is", sum)
```

5.1.2.5 Python break and continue

In this article, you will learn to use break and continue statements to alter the flow of a loop. In Python, break and continue statements can alter the flow of a normal loop. Loops iterate over a block of code until the test expression is false, but sometimes we wish to terminate the current iteration or even the whole loop without checking test expression.
The break and continue statements are used in these cases.

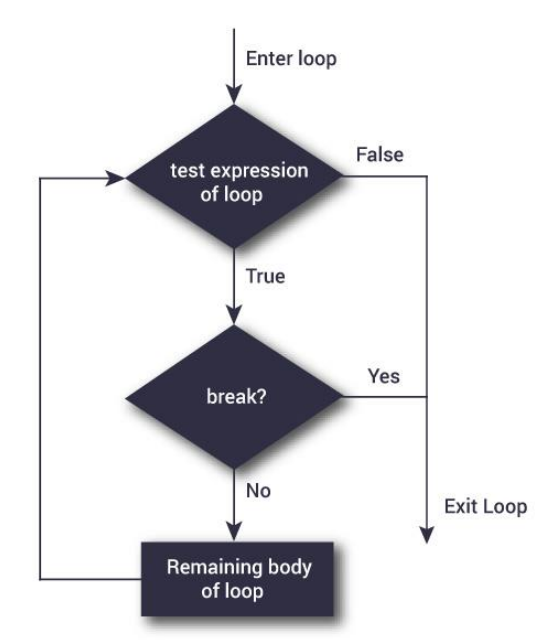
- **Python break statement**

The break statement terminates the loop containing it. Control of the program flows to the statement immediately after the body of the loop. If the break statement is inside a nested loop (loop inside another loop), the break statement will terminate the innermost loop.

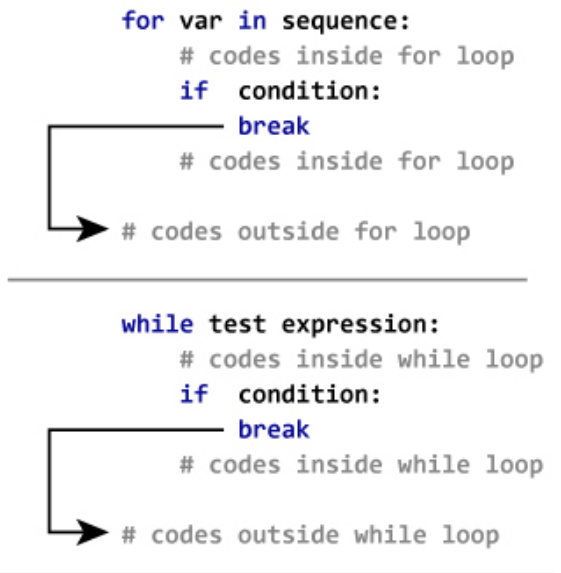
Syntax of break

```
break
```

Flowchart of break



The working of break statement in for loop and while loop is shown below.



Example: Python break

```
# Use of break statement inside the loop

for val in "string":
    if val == "i":
        break
    print(val)

print("The end")
```

In this program, we iterate through the "string" sequence. We check if the letter is i, upon which we break from the loop. Hence, we see in our output that all the letters up till i gets printed. After that, the loop terminates.

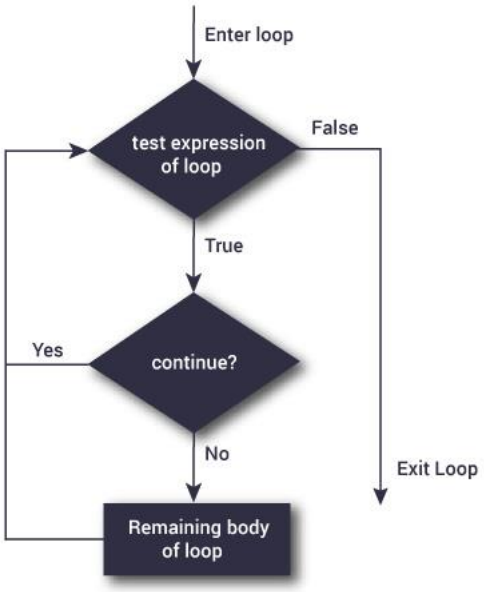
- **Python continue statement**

The continue statement is used to skip the rest of the code inside a loop for the current iteration only. Loop does not terminate but continues on with the next iteration.

Syntax of Continue

```
continue
```

Flowchart of continue



Flowchart of continue statement in Python

The working of continue statement in for and while loop is shown below.

```
for var in sequence:
    # codes inside for loop
    if  condition:
        continue
    # codes inside for loop

# codes outside for loop
```

```
while test expression:
    # codes inside while loop
    if  condition:
        continue
    # codes  inside while loop

# codes outside while loop
```

How continue statement works in python

Example: Python continue

```
# Program to show the use of continue statement inside loops

for val in "string":
    if val == "i":
        continue
    print(val)

print("The end")
```

This program is same as the above example except the break statement has been replaced with continue. We continue with the loop, if the string is i, not executing the rest of the block. Hence, we see in our output that all the letters except i gets printed.

5.1.2.6 Python pass statement

In Python programming, the pass statement is a null statement. The difference between a comment and a pass statement in Python is that while the interpreter ignores a comment entirely, pass is not ignored. However, nothing happens when the pass is executed. It results in no operation (NOP).

Syntax of pass

```
pass
```

We generally use it as a placeholder.

Suppose we have a loop or a function that is not implemented yet, but we want to implement it in the future. They cannot have an empty body. The interpreter would give an error. So, we use the pass statement to construct a body that does nothing.

Example: pass Statement

```
'''pass is just a placeholder for

functionality to be added later.'''

sequence = {'p', 'a', 's', 's'}

for val in sequence:

    pass
```

We can do the same thing in an empty function or class as well.

```
def function(args):

    pass

class Example:

    pass
```

5.1.3 PYTHON FUCTIONS

5.1.3.1 Function Syntax

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):
    """docstring"""
    statement(s)
```

Above shown is a function definition that consists of the following components.

Keyword `def` that marks the start of the function header.

A function name to uniquely identify the function. Function naming follows the same rules of writing identifiers in Python.

Parameters (arguments) through which we pass values to a function. They are optional.

A colon (:) to mark the end of the function header. Optional documentation string (docstring) to describe what the function does. One or more valid python statements that make up the function body. Statements must have the same indentation level (usually 4 spaces). An optional `return` statement to return a value from the function.

Example of a function

```
def greet(name):
    """
    This function greets to
    the person passed in as
    a parameter
    """
    print("Hello, " + name + ". Good morning!")
```

How to call a function in python?

Once we have defined a function, we can call it from another function, program or even the Python prompt. To call a function we simply type the function name with appropriate parameters.

```
>>> greet('Paul')
```

Hello, Paul. Good morning!

Note: Try running the above code in the Python program with the function definition to see the output.

5.1.3.2 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.

For example:

```
>>> print(greet("May"))
```

```
Hello, May. Good morning!None
```

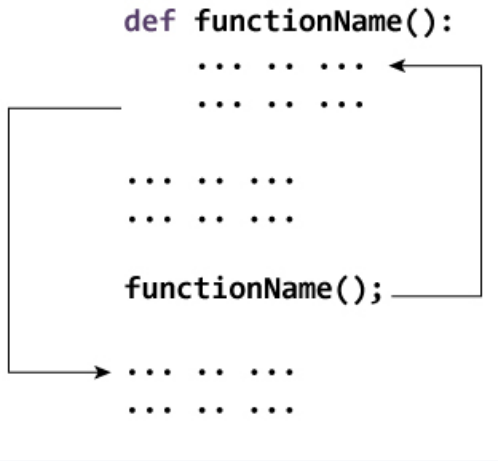
Example of return

```
def absolute_value(num):
    """This function returns the absolute
    value of the entered number"""

    if num >= 0:
        return num
    else:
        return -num

print(absolute_value(2))
print(absolute_value(-4))
```

Output

```
2
4
```

- **How Function works in Python?**



5.1.3.3 Scope and Lifetime of variables

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

Here, we can see that the value of x is 20 initially. Even though the function my_func() changed the value of x to 10, it did not affect the value outside the function. This is because the variable x inside the function is different (local to the function) from the one outside. Although they have the same names, they are two different variables with different scopes. On the other hand, variables outside of the function are visible from inside. They have a global scope. We can read these values from inside the function but cannot change (write) them. In order to modify the value of variables outside the function, they must be declared as global variables using the keyword global.

5.1.3.4 Types of Functions

Basically, we can divide functions into the following two types:

Built-in functions - Functions that are built into Python.

User-defined functions - Functions defined by the users themselves.

5.1.3.5 Python Function Arguments

In the user-defined function topic, we learned about defining a function and calling it. Otherwise, the function call will result in an error. Here is an example.

```
def greet(name, msg):
    """This function greets to
    the person with the provided message"""
    print("Hello", name + ', ' + msg)
greet("Monica", "Good morning!")
```

Output

```
Hello Monica, Good morning!
```

We can provide a default value to an argument by using the assignment operator (=). Here is an example.

```
def greet(name, msg="Good morning!"):
    print("Hello", name + ', ' + msg)
greet("Kate")
greet("Bruce", "How do you do?")
```

Output

```
Hello Kate, Good morning!
Hello Bruce, How do you do?
```

In this function, the parameter name does not have a default value and is required (mandatory) during a call. On the other hand, the parameter msg has a default value of "Good morning!". So, it is optional during a call. If a value is provided, it will overwrite the default value.

Any number of arguments in a function can have a default value. But once we have a default argument, all the arguments to its right must also have default values.

This means to say, non-default arguments cannot follow default arguments. For example, if we had defined the function header above as:

```
def greet(msg = "Good morning!", name):
```

We would get an error as:

----------Python Keyword Arguments

When we call a function with some values, these values get assigned to the arguments according to their position. For example, in the above function greet(), when we called it as greet("Bruce", "How do you do?"), the value "Bruce" gets assigned to the argument *name* and similarly "How do you do?" to *msg*.

----------Python Arbitrary Arguments

Sometimes, we do not know in advance the number of arguments that will be passed into a function. Python allows us to handle this kind of situation through function calls with an arbitrary number of arguments.

In the function definition, we use an asterisk (*) before the parameter name to denote this kind of argument. Here is an example.

```
def greet(*names):
    """This function greets all
    the person in the names tuple."""

    # names is a tuple with arguments
```

```
    for name in names:
        print("Hello", name)

greet("Monica", "Luke", "Steve", "John")
```

Here, we have called the function with multiple arguments. These arguments get wrapped up into a tuple before being passed into the function. Inside the function, we use a for loop to retrieve all the arguments back.

5.1.3.6 Python Variable Scope

Although there are various unique namespaces defined, we may not be able to access all of them from every part of the program. The concept of scope comes into play. A scope is the portion of a program from where a namespace can be accessed directly without any prefix.

If there is a function inside another function, a new scope is nested inside the local scope.

Example of Scope and Namespace in Python

```
def outer_function():
    b = 20
    def inner_func():
        c = 30
a = 10
```

Here, the variable *a* is in the global namespace. Variable *b* is in the local namespace of outer_function() and *c* is in the nested local namespace of inner_function(). When we are in inner_function(), *c* is local to us, *b* is nonlocal and *a* is global. We can read as well as assign new values to *c* but can only read *b* and *a* from inner_function().

If we try to assign as a value to *b*, a new variable *b* is created in the local namespace which is different than the nonlocal *b*. The same thing happens when we assign a value to *a*. However, if we declare *a* as global, all the reference and assignment go to the global *a*. Similarly, if we want to rebind the variable *b*, it must be declared as nonlocal. The following example will further clarify this.

```
def outer_function():
    a = 20
    def inner_function():
        a = 30
        print('a =', a)
    inner_function()
    print('a =', a)
a = 10
outer_function() print('a =', a)
```

As you can see, the output of this program is

a = 30

a = 20

a = 10

In this program, three different variables *a* are defined in separate namespaces and accessed accordingly. While in the following program,

```
def outer_function():
    global a
    a = 20
    def inner_function():
        global a
        a = 30
        print('a =', a)
    inner_function()
    print('a =', a)
a = 10
outer_function() print('a =', a)
```

The output of the program is.

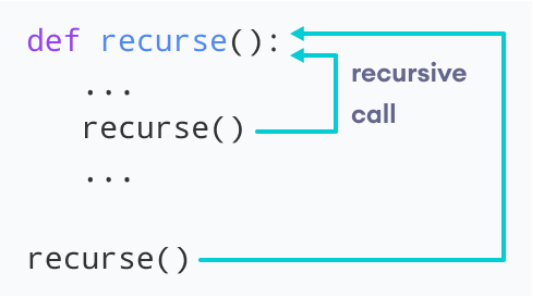
```
a = 30
a = 30
a = 30
```

Here, all references and assignments are to the global *a* due to the use of keyword global.

5.1.3.7 Python Recursive Function

Recursion is the process of defining something in terms of itself. A physical world example would be to place two parallel mirrors facing each other. Any object in between them would be reflected recursively. In Python, we know that a function can call other functions. It is even possible for the function to call itself. These types of construct are termed as recursive functions.

The following image shows the working of a recursive function called recurse.



Following is an example of a recursive function to find the factorial of an integer.

Factorial of a number is the product of all the integers from 1 to that number. For example, the factorial of 6 (denoted as 6!) is 1*2*3*4*5*6 = 720.

Example of a recursive function

```
def factorial(x):
    """This is a recursive function
    to find the factorial of an integer"""

    if x == 1:
        return 1
    else:
        return (x * factorial(x-1))
```

```
num = 3

print("The factorial of", num, "is", factorial(num))
```

Output

```
The factorial of 3 is 6
```

In the above example, factorial() is a recursive function as it calls itself. When we call this function with a positive integer, it will recursively call itself by decreasing the number. Each function multiplies the number with the factorial of the number below it until it is equal to one. This recursive call can be explained in the following steps.

```
factorial(3)          # 1st call with 3

3 * factorial(2)      # 2nd call with 2

3 * 2 * factorial(1)  # 3rd call with 1

3 * 2 * 1             # return from 3rd call as number=1

3 * 2                 # return from 2nd call

6                     # return from 1st call
```

Our recursion ends when the number reduces to 1. This is called the base condition. Every recursive function must have a base condition that stops the recursion or else the function calls itself infinitely.

5.1.3.8 Python Anonymous/Lambda Function

In Python, an anonymous function is a function that is defined without a name. While normal functions are defined using the def keyword in Python, anonymous functions are defined using the lambda keyword. Hence, anonymous functions are also called lambda functions.

A lambda function in python has the following syntax.

Syntax of Lambda Function in python

```
lambda arguments: expression
```

Lambda functions can have any number of arguments but only one expression. The expression is evaluated and returned. Lambda functions can be used wherever function objects are required.

Example of Lambda Function in python

Here is an example of lambda function that doubles the input value.

```
# Program to show the use of lambda functions

double = lambda x: x * 2

print(double(5))
```

Output

```
10
```

In the above program, `lambda x: x * 2` is the lambda function. Here *x* is the argument and `x * 2` is the expression that gets evaluated and returned. We can now call it as a normal function. The statement

```
double = lambda x: x * 2
```

is nearly the same as:

```
def double(x):
   return x * 2
```

- **Use of Lambda Function in python**

We use lambda functions when we require a nameless function for a short period of time. In Python, we generally use it as an argument to a higher-order function (a function that takes in other functions as arguments). Lambda functions are used along with built-in functions like filter(), map() etc.

Example use with filter()

The filter() function in Python takes in a function and a list as arguments. The function is called with all the items in the list and a new list is returned which contains items for which the function evaluates to True.

Here is an example use of filter() function to filter out only even numbers from a list.

```
# Program to filter out only the even items from a list

my_list = [1, 5, 4, 6, 8, 11, 3, 12]


new_list = list(filter(lambda x: (x%2 == 0) , my_list))

print(new_list)
```

Output

[4, 6, 8, 12]

Example use with map()

The map() function in Python takes in a function and a list. The function is called with all the items in the list and a new list is returned which contains items returned by that function for each item. Here is an example use of map() function to double all the items in a list.

```
 # Program to double each item in a list using map()
```

```
my_list = [1, 5, 4, 6, 8, 11, 3, 12]

new_list = list(map(lambda x: x * 2 , my_list))
print(new_list)
```

Output

```
[2, 10, 8, 12, 16, 22, 6, 24]
```

5.1.3.8 Python Modules

In this article, you will learn to create and import custom modules in Python. Also, you will find different techniques to import and use custom and built-in modules in Python. Modules refer to a file containing Python statements and definitions. A file containing Python code, for example: `example.py`, is called a module, and its module name would be `example`. We use modules to break down large programs into small manageable and organized files. Furthermore, modules provide reusability of code. We can define our most used functions in a module and import it, instead of copying their definitions into different programs.

Let us create a module. Type the following and save it as `example.py`.

```
def add(a, b):
   """This program adds two   numbers and return the result"""

   result = a + b
   return result
```

Here, we have defined a function add() inside a module named example. The function takes in two numbers and returns their sum.

----------How to import modules in Python?

We can import the definitions inside a module to another module or the interactive interpreter in Python. We use the import keyword to do this. To import our previously defined module example, we type the following in the Python prompt.

```
>>> import example
```

This does not import the names of the functions defined in example directly in the current symbol table. It only imports the module name example there.

Using the module name we can access the function using the dot . operator. For example:

```
>>> example.add(4,5.5)
9.5
```

----------Python import statement

We can import a module using the import statement and access the definitions inside it using the dot operator as described above. Here is an example.

```
# import statement example# to import standard module math
import math
print("The value of pi is", math.pi)
```

When you run the program, the output will be:

```
The value of pi is 3.141592653589793
```

----------Import with renaming

We can import a module by renaming it as follows:

```
# import module by renaming it

import math as m

print("The value of pi is", m.pi)
```

We have renamed the `math` module as `m`. This can save us typing time in some cases. Note that the name `math` is not recognized in our scope. Hence, `math.pi` is invalid, and `m.pi` is the correct implementation.

----------Python from...import statement

We can import specific names from a module without importing the module as a whole. Here is an example.

```
# import only pi from math module

from math import pi

print("The value of pi is", pi)
```

Here, we imported only the `pi` attribute from the `math` module. In such cases, we don't use the dot operator. We can also import multiple attributes as follows:

```
>>> from math import pi, e

>>> pi3.141592653589793

>>> e2.718281828459045
```

---------Import all names

We can import all names(definitions) from a module using the following construct:

```
# import all names from the standard module math

from math import *

print("The value of pi is", pi)
```

Here, we have imported all the definitions from the math module. This includes all names visible in our scope except those beginning with an underscore(private definitions). Importing everything with the asterisk (*) symbol is not a good programming practice. This can lead to duplicate definitions for an identifier. It also hampers the readability of our code.

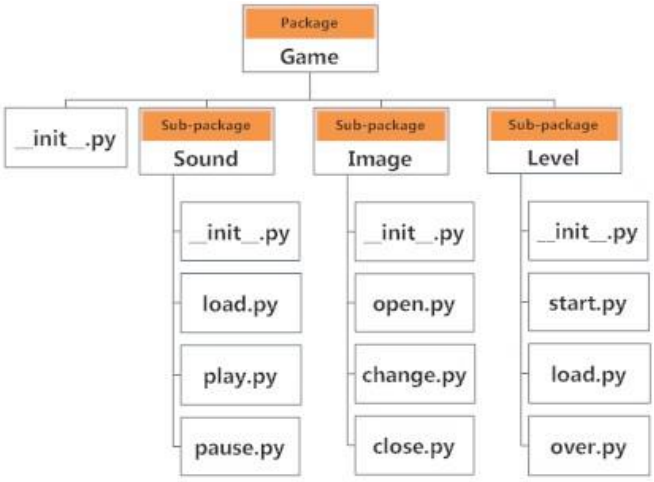
5.1.3.9 Python Package

We don't usually store all of our files on our computer in the same location. We use a well-organized hierarchy of directories for easier access. Similar files are kept in the same directory, for example, we may keep all the songs in the "music" directory. Analogous to this, Python has packages for directories and modules for files.

As our application program grows larger in size with a lot of modules, we place similar modules in one package and different modules in different packages. This makes a project (program) easy to manage and conceptually clear.

Similarly, as a directory can contain subdirectories and files, a Python package can have sub-packages and modules. A directory must contain a file named __init__.py in order for Python to consider it as a package. This file can be left empty but we generally place the initialization code for that package in this file.

Here is an example. Suppose we are developing a game. One possible organization of packages and modules could be as shown in the figure below.



.----------Importing module from a package

We can import modules from packages using the dot (.) operator. For example, if we want to import the start module in the above example, it can be done as follows:

import Game.Level.start

Now, if this module contains a function named select_difficulty(), we must use the full name to reference it.

Game.Level.start.select_difficulty(2)

If this construct seems lengthy, we can import the module without the package prefix as follows:

from Game.Level import start

We can now call the function simply as : >>>start.select_difficulty(2)

Another way of importing just the required function (or class or variable) from a module within a package would be as follows:

from Game.Level.start import select_difficulty

Now we can directly call this function. >>> select_difficulty(2)

5.1.4 PYTHON DATATYPES

5.1.4.1 Python List

In Python programming, a list is created by placing all the items (elements) inside square brackets [], separated by commas. It can have any number of items and they may be of different types (integer, float, string etc.).

```
# empty list
my_list = []
# list of integers
my_list = [1, 2, 3]
# list with mixed data types
my_list = [1, "Hello", 3.4]
```

A list can also have another list as an item. This is called a nested list.

```
# nested list
my_list = ["mouse", [8, 4, 6], ['a']]
```

We can use the index operator [] to access an item in a list. In Python, indices start at 0. So, a list having 5 elements will have an index from 0 to 4. Trying to access indexes other than these will raise an IndexError. The index must be an integer. We can't use float or other types, this will result in TypeError.

Nested lists are accessed using nested indexing.

```
# List indexing
```

```
my_list = ['p', 'r', 'o', 'b', 'e']

# Output: p

print(my_list[0])

# Output: o

print(my_list[2])

# Output: e

print(my_list[4])

# Nested List

n_list = ["Happy", [2, 0, 1, 5]]

# Nested indexing

print(n_list[0][1])

print(n_list[1][3])

# Error! Only integer can be used for indexingprint(my_list[4.0])
```

- **Negative indexing**

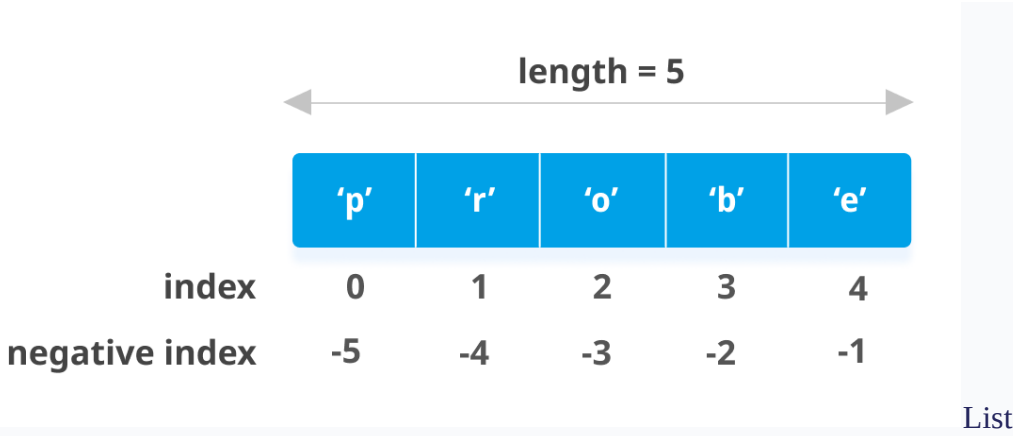
Python allows negative indexing for its sequences. The index of -1 refers to the last item, -2 to the second last item and so on.

```
# Negative indexing in lists

my_list = ['p','r','o','b','e']

print(my_list[-1])
```

```
print(my_list[-5])
```

When we run the above program, we will get the following output:

```
e

p
```



List

- **indexing in Python**

How to slice lists in Python?

We can access a range of items in a list by using the slicing operator :(colon).

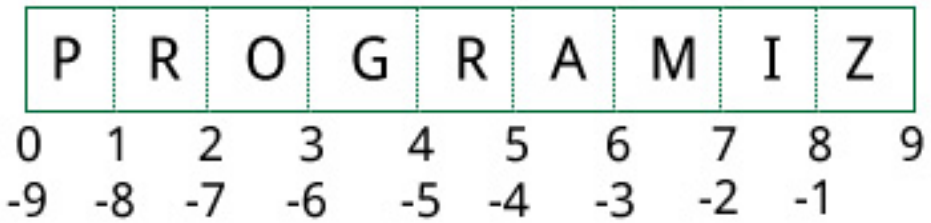
```
# List slicing in Python

my_list = ['p','r','o','g','r','a','m','i','z']

# elements 3rd to 5th

print(my_list[2:5])

# elements beginning to 4th

print(my_list[:-5])
```

```
# elements 6th to end

print(my_list[5:])

# elements beginning to end

print(my_list[:])
```

Output

```
['o', 'g', 'r']

['p', 'r', 'o', 'g']

['a', 'm', 'i', 'z']

['p', 'r', 'o', 'g', 'r', 'a', 'm', 'i', 'z']
```

Slicing can be best visualized by considering the index to be between the elements as shown below. So, if we want to access a range, we need two indices that will slice that portion from the list.



- **How to change or add elements to a list?**

Lists are mutable, meaning their elements can be changed unlike string or tuple. We can use the assignment operator (=) to change an item or a range of items.

```
# Correcting mistake values in a list

odd = [2, 4, 6, 8]
```

```
# change the 1st item

odd[0] = 1

print(odd)

# change 2nd to 4th items

odd[1:4] = [3, 5, 7]

print(odd)
```

We can add one item to a list using the append() method or add several items using extend() method.

```
# Appending and Extending lists in Python

odd = [1, 3, 5]

odd.append(7)

print(odd)

odd.extend([9, 11, 13])

print(odd)
```

Output

```
[1, 3, 5, 7]

[1, 3, 5, 7, 9, 11, 13]
```

We can also use + operator to combine two lists. This is also called concatenation.

The * operator repeats a list for the given number of times.

```
# Concatenating and repeating lists

odd = [1, 3, 5]
```

```
print(odd + [9, 7, 5])

print(["re"] * 3)
```

Output

```
[1, 3, 5, 9, 7, 5]

['re', 're', 're']
```

Furthermore, we can insert one item at a desired location by using the method insert() or insert multiple items by squeezing it into an empty slice of a list.

```
# Demonstration of list insert() method

odd = [1, 9]

odd.insert(1,3)

print(odd)


odd[2:2] = [5, 7]

print(odd)
```

Output

```
[1, 3, 9]

[1, 3, 5, 7, 9]
```

How to delete or remove elements from a list?

We can delete one or more items from a list using the keyword del. It can even delete the list entirely.

```
# Deleting list items
```

```
my_list = ['p', 'r', 'o', 'b', 'l', 'e', 'm']

# delete one item

del my_list[2]

print(my_list)

# delete multiple items

del my_list[1:5]

print(my_list)

# delete entire list

del my_list

# Error: List not defined

print(my_list)
```

Output

```
['p', 'r', 'b', 'l', 'e', 'm']

['p', 'm']

Traceback (most recent call last):

  File "<string>", line 18, in <module>

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

We can use `remove()` method to remove the given item or `pop()` method to remove an item at the given index. The `pop()` method removes and returns the last item if the index is not provided. This helps us implement lists as stacks (first in, last out data structure).

We can also use the `clear()` method to empty a list.

```
my_list = ['p','r','o','b','l','e','m']
```

```
my_list.remove('p')
# Output: ['r', 'o', 'b', 'l', 'e', 'm']
print(my_list)
# Output: 'o'
print(my_list.pop(1))
# Output: ['r', 'b', 'l', 'e', 'm']
print(my_list)
# Output: 'm'
print(my_list.pop())
# Output: ['r', 'b', 'l', 'e']
print(my_list)


my_list.clear()
# Output: []
print(my_list)
```

Output

```
['r', 'o', 'b', 'l', 'e', 'm']
o
['r', 'b', 'l', 'e', 'm']
m
['r', 'b', 'l', 'e']
```

```
[]
```

Finally, we can also delete items in a list by assigning an empty list to a slice of elements.

```
>>> my_list = ['p','r','o','b','l','e','m']
>>> my_list[2:3] = []
>>> my_list
['p', 'r', 'b', 'l', 'e', 'm']
>>> my_list[2:5] = []
>>> my_list
['p', 'r', 'm']
```

Python List Methods

Methods that are available with list objects in Python programming are tabulated below. They are accessed as list.method(). Some of the methods have already been used above.

Python List Methods

append() - Add an element to the end of the list

extend() - Add all elements of a list to the another list

insert() - Insert an item at the defined index

remove() - Removes an item from the list

pop() - Removes and returns an element at the given index

clear() - Removes all items from the list

index() - Returns the index of the first matched item

count() - Returns the count of the number of items passed as an argument

sort() - Sort items in a list in ascending order

reverse() - Reverse the order of items in the list

copy() - Returns a shallow copy of the list

Some examples of Python list methods:

```
# Python list methods
my_list = [3, 8, 1, 6, 0, 8, 4]
# Output: 1
print(my_list.index(8))
# Output: 2
print(my_list.count(8))
my_list.sort()
# Output: [0, 1, 3, 4, 6, 8, 8]
print(my_list)
my_list.reverse()
```

```
# Output: [8, 8, 6, 4, 3, 1, 0]

print(my_list)
```

Output

```
1

2

[0, 1, 3, 4, 6, 8, 8]

[8, 8, 6, 4, 3, 1, 0]
```

List Comprehension: Elegant way to create new List

List comprehension is an elegant and concise way to create a new list from an existing list in Python. A list comprehension consists of an expression followed by for statement inside square brackets. Here is an example to make a list with each item being increasing power of 2.

```
pow2 = [2 ** x for x in range(10)]

print(pow2)
```

Output

```
[1, 2, 4, 8, 16, 32, 64, 128, 256, 512]
```

This code is equivalent to:

```
pow2 = []

for x in range(10):

  pow2.append(2 ** x)
```

A list comprehension can optionally contain more for or if statements. An optional if statement can filter out items for the new list. Here are some examples.

```
>>> pow2 = [2 ** x for x in range(10) if x > 5]
>>> pow2
[64, 128, 256, 512]
>>> odd = [x for x in range(20) if x % 2 == 1]
>>> odd
[1, 3, 5, 7, 9, 11, 13, 15, 17, 19]
>>> [x+y for x in ['Python ','C '] for y in ['Language','Programming']]
['Python Language', 'Python Programming', 'C Language', 'C Programming']
```

Other List Operations in Python

List Membership Test

We can test if an item exists in a list or not, using the keyword in.

```
my_list = ['p', 'r', 'o', 'b', 'l', 'e', 'm']
# Output: True
print('p' in my_list)
# Output: False
print('a' in my_list)
# Output: True
print('c' not in my_list)
```

Iterating Through a List

Using a for loop we can iterate through each item in a list.

```
for fruit in ['apple','banana','mango']:
    print("I like",fruit)
```

5.1.4.2 Python Tuple

A tuple in Python is similar to a list. The difference between the two is that we cannot change the elements of a tuple once it is assigned whereas we can change the elements of a list.

----------Creating a Tuple

A tuple is created by placing all the items (elements) inside parentheses (), separated by commas. The parentheses are optional, however, it is a good practice to use them. A tuple can have any number of items and they may be of different types (integer, float, list, string, etc.).

```
# Different types of tuples

# Empty tuple

my_tuple = ()

print(my_tuple)

# Tuple having integers

my_tuple = (1, 2, 3)

print(my_tuple)

# tuple with mixed datatypes

my_tuple = (1, "Hello", 3.4)
```

```
print(my_tuple)

# nested tuple

my_tuple = ("mouse", [8, 4, 6], (1, 2, 3))

print(my_tuple)
```

A tuple can also be created without using parentheses. This is known as tuple packing.

```
my_tuple = 3, 4.6, "dog"

print(my_tuple)

# tuple unpacking is also possible

a, b, c = my_tuple

print(a)      # 3print(b)      # 4.6print(c)      # dog
```

----------Creating a tuple with one element is a bit tricky.

Having one element within parentheses is not enough. We will need a trailing comma to indicate that it is, in fact, a tuple.

```
my_tuple = ("hello") print(type(my_tuple))  # <class 'str'>

# Creating a tuple having one element

my_tuple = ("hello",) print(type(my_tuple))  # <class 'tuple'>

# Parentheses is optional

my_tuple = "hello", print(type(my_tuple))  # <class 'tuple'>
```

Output

```
<class 'str'>
```

<class 'tuple'>

<class 'tuple'>

----------Access Tuple Elements

There are various ways in which we can access the elements of a tuple.

1. Indexing

We can use the index operator [] to access an item in a tuple, where the index starts from 0.

So, a tuple having 6 elements will have indices from 0 to 5. Trying to access an index outside of the tuple index range(6,7,... in this example) will raise an IndexError. Likewise, nested tuples are accessed using nested indexing, as shown in the example below.

```
# Accessing tuple elements using indexing

my_tuple = ('p','e','r','m','i','t')

print(my_tuple[0])   # 'p' print(my_tuple[5])   # 't'

# IndexError: list index out of range# print(my_tuple[6])

# Index must be an integer# TypeError: list indices must be integers, not float# my_tuple[2.0]

# nested tuple

n_tuple = ("mouse", [8, 4, 6], (1, 2, 3))

# nested indexprint(n_tuple[0][3])      # 's'print(n_tuple[1][1])      # 4
```

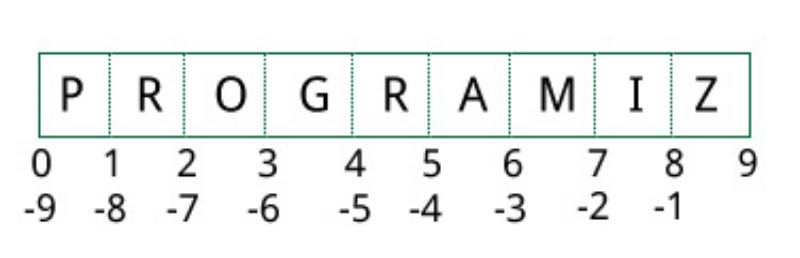
- **Slicing**

We can access a range of items in a tuple by using the slicing operator colon :.

```
# Accessing tuple elements using slicing

my_tuple = ('p','r','o','g','r','a','m','i','z')
```

```
# elements 2nd to 4th# Output: ('r', 'o', 'g')

print(my_tuple[1:4])

# elements beginning to 2nd# Output: ('p', 'r')

print(my_tuple[:-7])

# elements 8th to end# Output: ('i', 'z')

print(my_tuple[7:])

# elements beginning to end# Output: ('p', 'r', 'o', 'g', 'r', 'a', 'm', 'i', 'z')print(my_tuple[:])
```

Slicing can be best visualized by considering the index to be between the elements as shown below. So if we want to access a range, we need the index that will slice the portion from the tuple.



----------Changing a Tuple

Unlike lists, tuples are immutable.

This means that elements of a tuple cannot be changed once they have been assigned. But, if the element is itself a mutable data type like list, its nested items can be changed. We can also assign a tuple to different values (reassignment).

```
# Changing tuple values

my_tuple = (4, 2, 3, [6, 5])
```

```
# TypeError: 'tuple' object does not support item assignment# my_tuple[1] = 9

# However, item of mutable element can be changed

my_tuple[3][0] = 9    # Output: (4, 2, 3, [9, 5])

print(my_tuple)

# Tuples can be reassigned

my_tuple = ('p', 'r', 'o', 'g', 'r', 'a', 'm', 'i', 'z')

# Output: ('p', 'r', 'o', 'g', 'r', 'a', 'm', 'i', 'z')

print(my_tuple)
```

We can use + operator to combine two tuples. This is called concatenation.

We can also repeat the elements in a tuple for a given number of times using the * operator.

Both + and * operations result in a new tuple.

```
# Concatenation# Output: (1, 2, 3, 4, 5, 6)print((1, 2, 3) + (4, 5, 6))

# Repeat# Output: ('Repeat', 'Repeat', 'Repeat')print(("Repeat",) * 3)
```

Output

(1, 2, 3, 4, 5, 6)

('Repeat', 'Repeat', 'Repeat')

----------Deleting a Tuple

As discussed above, we cannot change the elements in a tuple. It means that we cannot delete or remove items from a tuple.

Deleting a tuple entirely, however, is possible using the keyword del.

```
# Deleting tuples

my_tuple = ('p', 'r', 'o', 'g', 'r', 'a', 'm', 'i', 'z')

# can't delete items# TypeError: 'tuple' object doesn't support item deletion# del my_tuple[3]

# Can delete an entire tuple

del my_tuple

# NameError: name 'my_tuple' is not defined

print(my_tuple)
```

Output

```
Traceback (most recent call last):

  File "<string>", line 12, in <module>

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

Tuple Methods

Methods that add items or remove items are not available with tuple. Only the following two methods are available.

Some examples of Python tuple methods:

```
my_tuple = ('a', 'p', 'p', 'l', 'e',)

print(my_tuple.count('p'))  # Output: 2

print(my_tuple.index('l'))  # Output: 3
```

----------Other Tuple Operations

1. Tuple Membership Test

We can test if an item exists in a tuple or not, using the keyword in.

```
# Membership test in tuple

my_tuple = ('a', 'p', 'p', 'l', 'e',)

# In operation

print('a' in my_tuple)print('b' in my_tuple)

# Not in operation

print('g' not in my_tuple)
```

Output

```
True

False

True
```

---

2. Iterating Through a Tuple

We can use a for loop to iterate through each item in a tuple.

```
# Using a for loop to iterate through a tuple

for name in ('John', 'Kate'):

    print("Hello", name)
```

Output

```
Hello John

Hello Kate
```

5.1.4.3 Python Strings

A string is a sequence of characters. A character is simply a symbol. For example, the English language has 26 characters. Computers do not deal with characters, they deal with numbers (binary). Even though you may see characters on your screen, internally it is stored and manipulated as a combination of 0s and 1s.

This conversion of character to a number is called encoding, and the reverse process is decoding. ASCII and Unicode are some of the popular encodings used. In Python, a string is a sequence of Unicode characters. Unicode was introduced to include every character in all languages and bring uniformity in encoding.

----------How to create a string in Python?

Strings can be created by enclosing characters inside a single quote or double-quotes. Even triple quotes can be used in Python but generally used to represent multiline strings and docstrings.

```
# defining strings in Python# all of the following are equivalent

my_string = 'Hello'

print(my_string)

my_string = "Hello"

print(my_string)

my_string = "'Hello"'

print(my_string)

# triple quotes string can extend multiple lines

my_string = """Hello, welcome to
```

```
        the world of Python"""
print(my_string)
```

----------How to access characters in a string?

We can access individual characters using indexing and a range of characters using slicing. Index starts from 0. The index of -1 refers to the last item, -2 to the second last item and so on. We can access a range of items in a string by using the slicing operator :(colon).

```
#Accessing string characters in Python
str = 'programiz'
print('str = ', str)
#first character
print('str[0] = ', str[0])
#last character
print('str[-1] = ', str[-1])
#slicing 2nd to 5th character
print('str[1:5] = ', str[1:5])
#slicing 6th to 2nd last character
print('str[5:-2] = ', str[5:-2])
```

When we run the above program, we get the following output:

```
str =  programiz
str[0] =  p
str[-1] =  z
```

str[1:5] = rogr

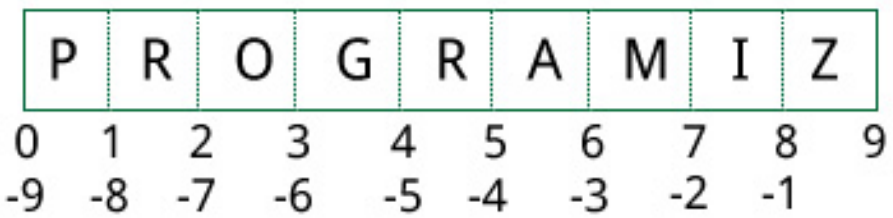
str[5:-2] = am

If we try to access an index out of the range or use numbers other than an integer, we will get errors.

```
# index must be in range

>>> my_string[15]

...

IndexError: string index out of range

# index must be an integer>>> my_string[1.5]

...

TypeError: string indices must be integers
```

Slicing can be best visualized by considering the index to be between the elements as shown below.

If we want to access a range, we need the index that will slice the portion from the string.



----------String Slicing in Python

How to change or delete a string?

Strings are immutable. This means that elements of a string cannot be changed once they have been assigned. We can simply reassign different strings to the same name.

```
>>> my_string = 'programiz'

>>> my_string[5] = 'a'

TypeError: 'str' object does not support item assignment>>> my_string = 'Python'>>> my_string'Python'
```

We cannot delete or remove characters from a string. But deleting the string entirely is possible using the `del` keyword.

```
>>> del my_string[1]

...

TypeError: 'str' object doesn't support item deletion

>>> del my_string

>>> my_string

...

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

----------Python String Operations

There are many operations that can be performed with strings which makes it one of the most used data types in Python.

1. Concatenation of Two or More Strings

Joining of two or more strings into a single one is called concatenation.

The + operator does this in Python. Simply writing two string literals together also concatenates them.

The * operator can be used to repeat the string for a given number of times.

```
# Python String Operations

str1 = 'Hello'

str2 ='World!'

# using +print('str1 + str2 = ', str1 + str2)

# using *print('str1 * 3 =', str1 * 3)
```

When we run the above program, we get the following output:

```
str1 + str2 =  HelloWorld!

str1 * 3 = HelloHelloHello
```

Writing two string literals together also concatenates them like + operator.

If we want to concatenate strings in different lines, we can use parentheses.

```
>>> # two string literals together>>> 'Hello "World!"Hello World!'

>>> # using parentheses>>> s = ('Hello '...      'World')

>>> s

'Hello World'
```

----------Iterating Through a string

We can iterate through a string using a for loop. Here is an example to count the number of 'l's in a string.

```
# Iterating through a string

count = 0

for letter in 'Hello World':

   if(letter == 'l'):

      count += 1

print(count,'letters found')
```

----------String Membership Test

We can test if a substring exists within a string or not, using the keyword in.

```
>>> 'a' in 'program'

True

>>> 'at' not in 'battle'

False
```

----------Built-in functions to Work with Python

Various built-in functions that work with sequence work with strings as well.

Some of the commonly used ones are enumerate() and len(). The enumerate() function returns an enumerate object. It contains the index and value of all the items in the string as pairs. This can be useful for iteration.

Similarly, len() returns the length (number of characters) of the string.

```
str = 'cold'

# enumerate()

list_enumerate = list(enumerate(str))

print('list(enumerate(str) = ', list_enumerate)

#character count

print('len(str) = ', len(str))
```

When we run the above program, we get the following output:

```
list(enumerate(str) =  [(0, 'c'), (1, 'o'), (2, 'l'), (3, 'd')]

len(str) =  4
```

----------Common Python String Methods

There are numerous methods available with the string object. The format() method that we mentioned above is one of them. Some of the commonly used methods are lower(), upper(), join(), split(), find(), replace() etc. Here is a complete list of all the built-in methods to work with strings in Python.

```
>>> "PrOgRaMiZ".lower()

'programiz'

>>> "PrOgRaMiZ".upper()

'PROGRAMIZ'

>>> "This will split all words into a list".split()

['This', 'will', 'split', 'all', 'words', 'into', 'a', 'list']
```

```
>>> ' '.join(['This', 'will', 'join', 'all', 'words', 'into', 'a', 'string'])
'This will join all words into a string'
>>> 'Happy New Year'.find('ew')
 7
>>> 'Happy New Year'.replace('Happy','Brilliant')
'Brilliant New Year'
```

5.1.4.4 Python Sets

A set is an unordered collection of items. Every set element is unique (no duplicates) and must be immutable (cannot be changed). However, a set itself is mutable. We can add or remove items from it. Sets can also be used to perform mathematical set operations like union, intersection, symmetric difference, etc.

----------Creating Python Sets

A set is created by placing all the items (elements) inside curly braces {}, separated by comma, or by using the built-in set() function. It can have any number of items and they may be of different types (integer, float, tuple, string etc.). But a set cannot have mutable elements like lists, sets or dictionaries as its elements.

```
# Different types of sets in Python# set of integers
my_set = {1, 2, 3}
print(my_set)
```

```
# set of mixed datatypes
my_set = {1.0, "Hello", (1, 2, 3)}
print(my_set)
```

Output

```
{1, 2, 3}
{1.0, (1, 2, 3), 'Hello'}
```

----------Creating an empty set is a bit tricky.

Empty curly braces `{}` will make an empty dictionary in Python. To make a set without any elements, we use the `set()` function without any argument.

```
# Distinguish set and dictionary while creating empty set
# initialize a with {}
a = {}
# check data type of a
print(type(a))
# initialize a with set()
a = set()
# check data type of a
print(type(a))
```

Output

```
<class 'dict'>
```

<class 'set'>

----------Modifying a set in Python

Sets are mutable. However, since they are unordered, indexing has no meaning.

We cannot access or change an element of a set using indexing or slicing. Set data type does not support it. We can add a single element using the add() method, and multiple elements using the update() method. The update() method can take tuples, lists, strings or other sets as its argument. In all cases, duplicates are avoided.

```
# initialize my_set

my_set = {1, 3}

print(my_set)

# if you uncomment line 9,# you will get an error# TypeError: 'set' object does not support indexing

# my_set[0]

# add an element# Output: {1, 2, 3}

my_set.add(2)

print(my_set)

# add multiple elements# Output: {1, 2, 3, 4}

my_set.update([2, 3, 4])

print(my_set)

# add list and set# Output: {1, 2, 3, 4, 5, 6, 8}

my_set.update([4, 5], {1, 6, 8})
```

```
print(my_set)
```

Output

```
{1, 3}

{1, 2, 3}

{1, 2, 3, 4}

{1, 2, 3, 4, 5, 6, 8}
```

----------Removing elements from a set

A particular item can be removed from a set using the methods `discard()` and `remove()`.

The only difference between the two is that the `discard()` function leaves a set unchanged if the element is not present in the set. On the other hand, the `remove()` function will raise an error in such a condition (if element is not present in the set).

The following example will illustrate this.

```
# Difference between discard() and remove()

# initialize my_set

my_set = {1, 3, 4, 5, 6}

print(my_set)

# discard an element# Output: {1, 3, 5, 6}

my_set.discard(4)

print(my_set)

# remove an element# Output: {1, 3, 5}

my_set.remove(6)
```

```
print(my_set)

# discard an element# not present in my_set# Output: {1, 3, 5}

my_set.discard(2)

print(my_set)

# remove an element# not present in my_set# you will get an error.# Output: KeyError


my_set.remove(2)
```

Output

```
{1, 3, 4, 5, 6}

{1, 3, 5, 6}

{1, 3, 5}

{1, 3, 5}

Traceback (most recent call last):

  File "<string>", line 28, in <module>

KeyError: 2
```

Similarly, we can remove and return an item using the `pop()` method.

Since set is an unordered data type, there is no way of determining which item will be popped. It is completely arbitrary.

We can also remove all the items from a set using the `clear()` method.

```
# initialize my_set# Output: set of unique elements

my_set = set("HelloWorld")
```

```
print(my_set)

# pop an element# Output: random element

print(my_set.pop())

# pop another element

my_set.pop()

print(my_set)

# clear my_set# Output: set()

my_set.clear()p

rint(my_set)

print(my_set)
```

Output

```
{'H', 'l', 'r', 'W', 'o', 'd', 'e'}

H

{'r', 'W', 'o', 'd', 'e'}

set()
```

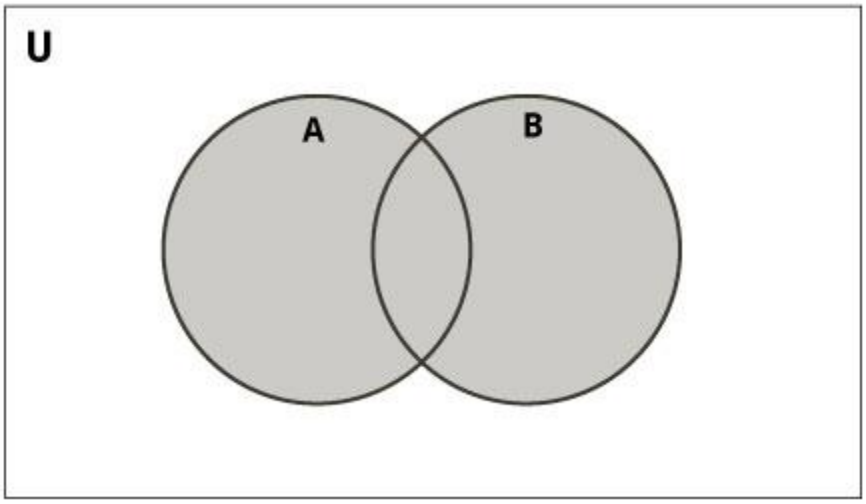
----------Python Set Operations

Sets can be used to carry out mathematical set operations like union, intersection, difference and symmetric difference. We can do this with operators or methods.

Let us consider the following two sets for the following operations.

```
>>> A = {1, 2, 3, 4, 5}>>> B = {4, 5, 6, 7, 8}
```

Set Union



Set Union in Python

Union of *A* and *B* is a set of all elements from both sets.

Union is performed using | operator. Same can be accomplished using the union() method.

```
# Set union method# initialize A and B

A = {1, 2, 3, 4, 5}

B = {4, 5, 6, 7, 8}

# use | operator# Output: {1, 2, 3, 4, 5, 6, 7, 8}

print(A | B)
```

Output

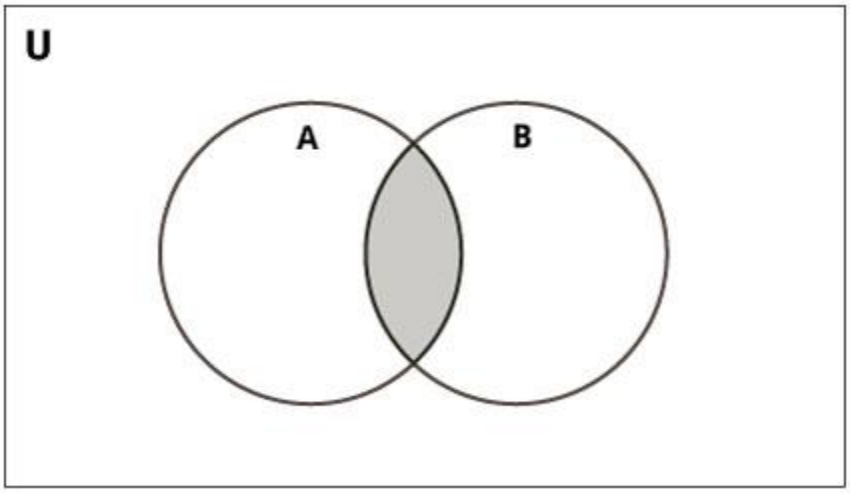
```
{1, 2, 3, 4, 5, 6, 7, 8}
```

Try the following examples on Python shell.

```
# use union function

>>> A.union(B)

{1, 2, 3, 4, 5, 6, 7, 8}
```

```
# use union function on B

>>> B.union(A)

{1, 2, 3, 4, 5, 6, 7, 8}
```

Set Intersection



----------Set Intersection in Python

Intersection of *A* and *B* is a set of elements that are common in both the sets.

Intersection is performed using & operator. Same can be accomplished using the intersection() method.

```
# Intersection of sets# initialize A and B

A = {1, 2, 3, 4, 5}

B = {4, 5, 6, 7, 8}

# use & operator# Output: {4, 5}

print(A & B)
```

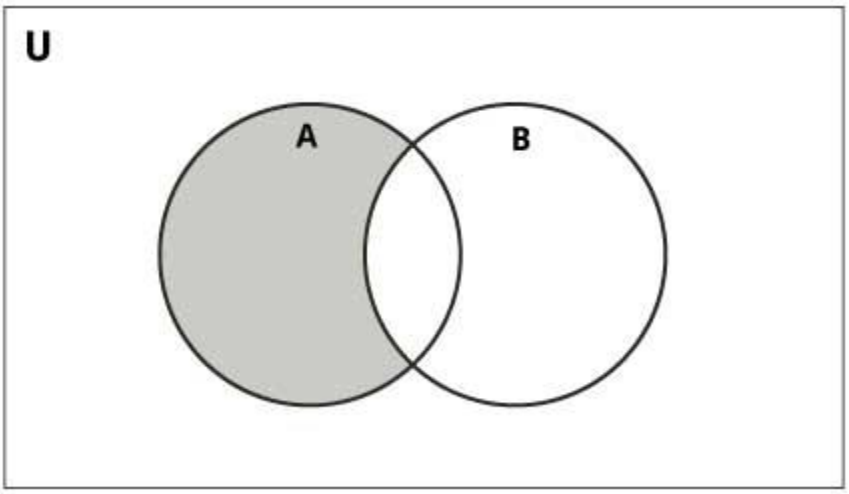
Output

{4, 5}

Try the following examples on Python shell.

```
# use intersection function on A

>>> A.intersection(B)

{4, 5}

# use intersection function on B>>>

B.intersection(A)

{4, 5}
```

Set Difference



----------Set Difference in Python

Difference of the set B from set A(A - B) is a set of elements that are only in A but not in B. Similarly, B - A is a set of elements in B but not in A.

Difference is performed using - operator. Same can be accomplished using the difference() method.

```
# Difference of two sets# initialize A and B

A = {1, 2, 3, 4, 5}

B = {4, 5, 6, 7, 8}

# use - operator on A# Output: {1, 2, 3}

print(A - B)
```

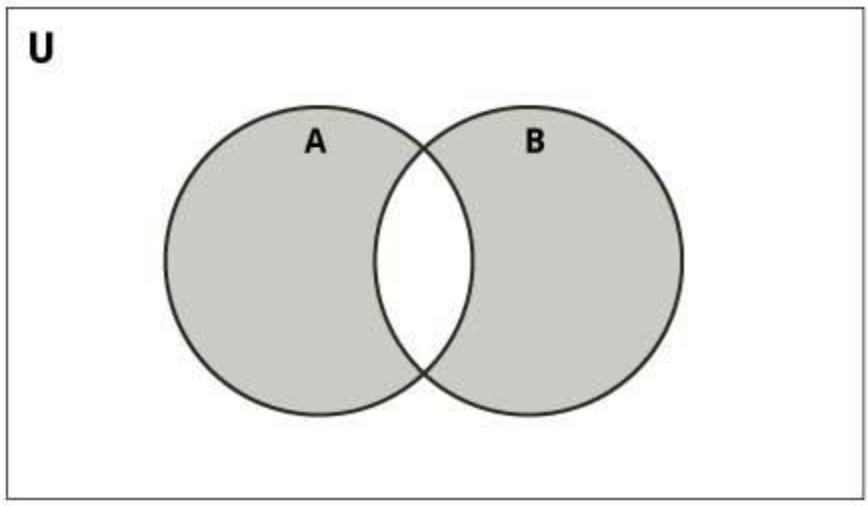
Output

```
{1, 2, 3}
```

Try the following examples on Python shell.

```
# use difference function on A>>>

A.difference(B)

{1, 2, 3}

# use - operator on B>>> B - A

{8, 6, 7}

# use difference function on B>>>

B.difference(A)

{8, 6, 7}
```

Set Symmetric Difference



----------Set Symmetric Difference in Python

Symmetric Difference of *A* and *B* is a set of elements in *A* and *B* but not in both (excluding the intersection).

Symmetric difference is performed using ^ operator. Same can be accomplished using the method symmetric_difference().

```
# Symmetric difference of two sets# initialize A and B

A = {1, 2, 3, 4, 5}

B = {4, 5, 6, 7, 8}

# use ^ operator# Output: {1, 2, 3, 6, 7, 8}

print(A ^ B)
```

Output

```
{1, 2, 3, 6, 7, 8}
```

----------Other Python Set Methods

There are many set methods, some of which we have already used above. Here is a list of all the methods that are available with the set objects:

Method	Description
add()	Adds an element to the set
clear()	Removes all elements from the set
copy()	Returns a copy of the set
difference()	Returns the difference of two or more sets as a new set
difference_update()	Removes all elements of another set from this set
discard()	Removes an element from the set if it is a member. (Do nothing if the element is not in set)
intersection()	Returns the intersection of two sets as a new set
intersection_update()	Updates the set with the intersection of itself and another
isdisjoint()	Returns True if two sets have a null intersection

issubset()	Returns True if another set contains this set
issuperset()	Returns True if this set contains another set
pop()	Removes and returns an arbitrary set element. Raises KeyError if the set is empty
remove()	Removes an element from the set. If the element is not a member, raises a KeyError
symmetric_difference()	Returns the symmetric difference of two sets as a new set
symmetric_difference_update()	Updates a set with the symmetric difference of itself and another
union()	Returns the union of sets in a new set
update()	Updates the set with the union of itself and others

----------Other Set Operations

Set Membership Test

We can test if an item exists in a set or not, using the in keyword.

```
# in keyword in a set# initialize my_set

my_set = set("apple")
```

```
# check if 'a' is present# Output: True
print('a' in my_set)
# check if 'p' is present# Output: False
print('p' not in my_set)
```

Output

```
True
False
```

----------Iterating Through a Set

We can iterate through each item in a set using a `for` loop.

```
>>> for letter in set("apple"):...     print(letter)...
a
p
e
l
```

----------Built-in Functions with Set

Built-in functions like `all()`, `any()`, `enumerate()`, `len()`, `max()`, `min()`, `sorted()`, `sum()` etc. are commonly used with sets to perform different tasks.

Function	Description

all()	Returns True if all elements of the set are true (or if the set is empty).
any()	Returns True if any element of the set is true. If the set is empty, returns False.
enumerate()	Returns an enumerate object. It contains the index and value for all the items of the set as a pair.
len()	Returns the length (the number of items) in the set.
max()	Returns the largest item in the set.
min()	Returns the smallest item in the set.
sorted()	Returns a new sorted list from elements in the set(does not sort the set itself).
sum()	Returns the sum of all elements in the set.

5.1.4.5 Python Dictionary

Python dictionary is an unordered collection of items. Each item of a dictionary has a key/value pair. Dictionaries are optimized to retrieve values when the key is known.

----------Creating Python Dictionary

Creating a dictionary is as simple as placing items inside curly braces {} separated by commas.

An item has a key and a corresponding value that is expressed as a pair (key: value).

While the values can be of any data type and can repeat, keys must be of immutable type (string, number or tuple with immutable elements) and must be unique.

```
# empty dictionary

my_dict = {}

# dictionary with integer keys

my_dict = {1: 'apple', 2: 'ball'}

# dictionary with mixed keys

my_dict = {'name': 'John', 1: [2, 4, 3]}

# using dict()

my_dict = dict({1:'apple', 2:'ball'})

# from sequence having each item as a pair

my_dict = dict([(1,'apple'), (2,'ball')])
```

As you can see from above, we can also create a dictionary using the built-in dict() function.

----------Accessing Elements from Dictionary

While indexing is used with other data types to access values, a dictionary uses keys. Keys can be used either inside square brackets [] or with the get() method. If we use the square brackets [], KeyError is raised in case a key is not found in the dictionary. On the other hand, the get() method returns None if the key is not found.

```
# get vs [] for retrieving elements

my_dict = {'name': 'Jack', 'age': 26}

# Output: Jack

print(my_dict['name'])

# Output: 26

print(my_dict.get('age'))

# Trying to access keys which doesn't exist throws error# Output None

print(my_dict.get('address'))

# KeyError

print(my_dict['address'])
```

----------Changing and Adding Dictionary elements

Dictionaries are mutable. We can add new items or change the value of existing items using an assignment operator.

If the key is already present, then the existing value gets updated. In case the key is not present, a new (key: value) pair is added to the dictionary.

```
# Changing and adding Dictionary Elements

my_dict = {'name': 'Jack', 'age': 26}

# update value
```

```
my_dict['age'] = 27

#Output: {'age': 27, 'name': 'Jack'}

print(my_dict)

# add item

my_dict['address'] = 'Downtown'

# Output: {'address': 'Downtown', 'age': 27, 'name': 'Jack'}

print(my_dict)
```

Output

```
{'name': 'Jack', 'age': 27}

{'name': 'Jack', 'age': 27, 'address': 'Downtown'}
```

----------Removing elements from Dictionary

We can remove a particular item in a dictionary by using the pop() method. This method removes an item with the provided key and returns the value. The popitem() method can be used to remove and return an arbitrary (key, value) item pair from the dictionary. All the items can be removed at once, using the clear() method.

We can also use the del keyword to remove individual items or the entire dictionary itself.

```
# Removing elements from a dictionary

# create a dictionary

squares = {1: 1, 2: 4, 3: 9, 4: 16, 5: 25}

# remove a particular item, returns its value# Output: 16

print(squares.pop(4))
```

```
# Output: {1: 1, 2: 4, 3: 9, 5: 25}print(squares)

# remove an arbitrary item, return (key,value)# Output: (5, 25)

print(squares.popitem())

# Output: {1: 1, 2: 4, 3: 9}

print(squares)

# remove all items

squares.clear()

# Output: {}

print(squares)

# delete the dictionary itself

del squares

# Throws Error

print(squares)
```

Output

```
16

{1: 1, 2: 4, 3: 9, 5: 25}

(5, 25)

{1: 1, 2: 4, 3: 9}

{}

Traceback (most recent call last):

  File "<string>", line 30, in <module>
```

```
print(squares)
```

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

----------Python Dictionary Methods

Methods that are available with a dictionary are tabulated below. Some of them have already been used in the above examples.

Method	Description
clear()	Removes all items from the dictionary.
copy()	Returns a shallow copy of the dictionary.
fromkeys(seq[, v])	Returns a new dictionary with keys from *seq* and value equal to *v* (defaults to None).
get(key[,d])	Returns the value of the *key*. If the *key* does not exist, returns *d* (defaults to None).
items()	Return a new object of the dictionary's items in (key, value) format.
keys()	Returns a new object of the dictionary's keys.
pop(key[,d])	Removes the item with the *key* and returns its value or *d* if *key* is not found. If *d* is not provided and the *key* is not found, it raises KeyError.

popitem()	Removes and returns an arbitrary item (key, value). Raises KeyError if the dictionary is empty.
setdefault(key[,d])	Returns the corresponding value if the *key* is in the dictionary. If not, inserts the *key* with a value of *d* and returns *d* (defaults to None).
update([other])	Updates the dictionary with the key/value pairs from *other*, overwriting existing keys.
values()	Returns a new object of the dictionary's values

Here are a few example use cases of these methods.

```
# Dictionary Methods

marks = {}.fromkeys(['Math', 'English', 'Science'], 0)

# Output: {'English': 0, 'Math': 0, 'Science': 0}

print(marks)

for item in marks.items():

   print(item)

# Output: ['English', 'Math', 'Science']

print(list(sorted(marks.keys())))
```

Output

```
{'Math': 0, 'English': 0, 'Science': 0}

('Math', 0)
```

('English', 0)

('Science', 0)

['English', 'Math', 'Science']

----------Python Dictionary Comprehension

Dictionary comprehension is an elegant and concise way to create a new dictionary from an iterable in Python. Dictionary comprehension consists of an expression pair (key: value) followed by a for statement inside curly braces {}. Here is an example to make a dictionary with each item being a pair of a number and its square.

```
# Dictionary Comprehension

squares = {x: x*x for x in range(6)}

print(squares)
```

Output

{0: 0, 1: 1, 2: 4, 3: 9, 4: 16, 5: 25}

This code is equivalent to

```
squares = {}

for x in range(6):

   squares[x] = x*x

print(squares)
```

Output

{0: 0, 1: 1, 2: 4, 3: 9, 4: 16, 5: 25}

A dictionary comprehension can optionally contain more for or if statements.

An optional if statement can filter out items to form the new dictionary. Here are some examples to make a dictionary with only odd items.

```
# Dictionary Comprehension with if conditional

odd_squares = {x: x*x for x in range(11) if x % 2 == 1}

print(odd_squares)
```

Output

```
{1: 1, 3: 9, 5: 25, 7: 49, 9: 81}
```

----------Other Dictionary Operations

Dictionary Membership Test

We can test if a key is in a dictionary or not using the keyword in. Notice that the membership test is only for the keys and not for the values.

```
# Membership Test for Dictionary Keys

squares = {1: 1, 3: 9, 5: 25, 7: 49, 9: 81}

# Output: True

print(1 in squares)

# Output: True

print(2 not in squares)

# membership tests for key only not value# Output: False

print(49 in squares)
```

Output

```
True
```

True

False

----------**Iterating Through a Dictionary**

We can iterate through each key in a dictionary using a for loop.

```
# Iterating through a Dictionary

squares = {1: 1, 3: 9, 5: 25, 7: 49, 9: 81}

for i in squares:

    print(squares[i])
```

Output

1

9

25

49

81

5.1.5 PYTHON FILES

In this tutorial, you'll learn about Python file operations. More specifically, opening a file, reading from it, writing into it, closing it, and various file methods that you should be aware of.

Files are named locations on disk to store related information. They are used to permanently store data in a non-volatile memory (e.g. hard disk). Since Random Access Memory (RAM) is volatile (which loses its data when the computer is turned off), we use files for future use of the data by permanently storing them.

When we want to read from or write to a file, we need to open it first. When we are done, it needs to be closed so that the resources that are tied with the file are freed. Hence, in Python, a file operation takes place in the following order:

5.1.5.1 Open a file

Read or write (perform operation)

Close the file

----------Opening Files in Python

Python has a built-in open() function to open a file. This function returns a file object, also called a handle, as it is used to read or modify the file accordingly.

```
>>> f = open("test.txt")    # open file in current directory>>> f =
open("C:/Python38/README.txt")  # specifying full path
```

We can specify the mode while opening a file. In mode, we specify whether we want to read r, write w or append a to the file. We can also specify if we want to open the file in text mode or binary mode. The default is reading in text mode. In this mode, we get strings when reading from the file.

On the other hand, binary mode returns bytes and this is the mode to be used when dealing with non-text files like images or executable files.

Mode	Description
r	Opens a file for reading. (default)
w	Opens a file for writing. Creates a new file if it does not exist or truncates the file if it exists.
x	Opens a file for exclusive creation. If the file already exists, the operation fails.
a	Opens a file for appending at the end of the file without truncating it. Creates a new file if it does not exist.
t	Opens in text mode. (default)
b	Opens in binary mode.
+	Opens a file for updating (reading and writing)

```
f = open("test.txt")      # equivalent to 'r' or 'rt'
f = open("test.txt",'w')  # write in text mode
f = open("img.bmp",'r+b') # read and write in binary mode
```

Unlike other languages, the character `a` does not imply the number 97 until it is encoded using `ASCII` (or other equivalent encodings).

Moreover, the default encoding is platform dependent. In windows, it is `cp1252` but `utf-8` in Linux. So, we must not also rely on the default encoding or else our code will behave differently in different platforms.

Hence, when working with files in text mode, it is highly recommended to specify the encoding type.

```
f = open("test.txt", mode='r', encoding='utf-8')
```

----------Closing Files in Python

When we are done with performing operations on the file, we need to properly close the file.

Closing a file will free up the resources that were tied with the file. It is done using the `close()` method available in Python. Python has a garbage collector to clean up unreferenced objects but we must not rely on it to close the file.

```
f = open("test.txt", encoding = 'utf-8')# perform file operations
f.close()
```

This method is not entirely safe. If an exception occurs when we are performing some operation with the file, the code exits without closing the file. A safer way is to use a try...finally block.

```
try:
  f = open("test.txt", encoding = 'utf-8')
  # perform file operationsfinally:
  f.close()
```

This way, we are guaranteeing that the file is properly closed even if an exception is raised that causes program flow to stop. The best way to close a file is by using the `with` statement. This ensures that the file is closed when the block inside the `with` statement is exited.

We don't need to explicitly call the `close()` method. It is done internally.

```
with open("test.txt", encoding = 'utf-8') as f:
  # perform file operations
```

5.1.5.2 Writing to Files in Python

In order to write into a file in Python, we need to open it in write `w`, append `a` or exclusive creation `x` mode. We need to be careful with the `w` mode, as it will overwrite into the file if it already exists. Due to this, all the previous data are erased. Writing a string or sequence of bytes (for binary files) is done using the `write()` method. This method returns the number of characters written to the file.

```
with open("test.txt",'w',encoding = 'utf-8') as f:
  f.write("my first file\n")
  f.write("This file\n\n")
  f.write("contains three lines\n")
```

This program will create a new file named `test.txt` in the current directory if it does not exist. If it does exist, it is overwritten. We must include the newline characters ourselves to distinguish the different lines.

5.1.5.3 Reading Files in Python

To read a file in Python, we must open the file in reading `r` mode. There are various methods available for this purpose. We can use the `read(size)` method to read in the *size* number of data. If the *size* parameter is not specified, it reads and returns up to the end of the file. We can read the `text.txt` file we wrote in the above section in the following way:

```
>>> f = open("test.txt",'r',encoding = 'utf-8')>>> f.read(4)    # read the first 4 data'This'
>>> f.read(4)    # read the next 4 data' is '
>>> f.read()     # read in the rest till end of file'my first file\nThis file\ncontains three lines\n'
```

```
>>> f.read()  # further reading returns empty sting"
```

We can see that the read() method returns a newline as '\n'. Once the end of the file is reached, we get an empty string on further reading. We can change our current file cursor (position) using the seek() method. Similarly, the tell() method returns our current position (in number of bytes).

```
>>> f.tell()    # get the current file position56
>>> f.seek(0)   # bring file cursor to initial position0
>>> print(f.read())  # read the entire file
This is my first file
This file
contains three lines
```

We can read a file line-by-line using a for loop. This is both efficient and fast.

```
>>> for line in f:...     print(line, end = '')
...
This is my first file
This file
contains three lines
```

In this program, the lines in the file itself include a newline character \n. So, we use the end parameter of the print() function to avoid two newlines when printing. Alternatively, we can use the readline() method to read individual lines of a file. This method reads a file till the newline, including the newline character.

```
>>> f.readline()'This is my first file\n'
>>> f.readline()'This file\n'
>>> f.readline()'contains three lines\n'
```

```
>>> f.readline()"
```

Lastly, the `readlines()` method returns a list of remaining lines of the entire file. All these reading methods return empty values when the end of file (EOF) is reached.

```
>>> f.readlines()
['This is my first file\n', 'This file\n', 'contains three lines\n']
```

Python File Methods

There are various methods available with the file object. Some of them have been used in the above examples.

Here is the complete list of methods in text mode with a brief description:

Method	Description
close()	Closes an opened file. It has no effect if the file is already closed.
detach()	Separates the underlying binary buffer from the `TextIOBase` and returns it.
fileno()	Returns an integer number (file descriptor) of the file.
flush()	Flushes the write buffer of the file stream.
isatty()	Returns `True` if the file stream is interactive.

read(*n*)	Reads at most *n* characters from the file. Reads till end of file if it is negative or None.
readable()	Returns True if the file stream can be read from.
readline(*n*=-1)	Reads and returns one line from the file. Reads in at most *n* bytes if specified.
readlines(*n*=-1)	Reads and returns a list of lines from the file. Reads in at most *n* bytes/characters if specified.
seek(*offset*,*from*=SEEK_SET)	Changes the file position to *offset* bytes, in reference to *from* (start, current, end).
seekable()	Returns True if the file stream supports random access.
tell()	Returns the current file location.
truncate(*size*=None)	Resizes the file stream to *size* bytes. If *size* is not specified, resizes to current location.
writable()	Returns True if the file stream can be written to.
write(*s*)	Writes the string *s* to the file and returns the number of characters written.
writelines(*lines*)	Writes a list of *lines* to the file.

5.1.5.4 Python Directory and Files Management

In this tutorial, you'll learn about file and directory management in Python, i.e. creating a directory, renaming it, listing all directories, and working with them.

-----Python Directory

If there are a large number of files to handle in our Python program, we can arrange our code within different directories to make things more manageable. A directory or folder is a collection of files and subdirectories. Python has the os module that provides us with many useful methods to work with directories (and files as well).

--------Get Current Directory

We can get the present working directory using the getcwd() method of the os module. This method returns the current working directory in the form of a string. We can also use the getcwdb() method to get it as bytes object.

```
>>> import os
>>> os.getcwd()'C:\\Program Files\\PyScripter'
>>> os.getcwdb()b'C:\\Program Files\\PyScripter'
```

The extra backslash implies an escape sequence. The print() function will render this properly.

```
>>> print(os.getcwd())
C:\Program Files\PyScripter
```

----------Changing Directory

We can change the current working directory by using the chdir() method.

The new path that we want to change into must be supplied as a string to this method. We can use both the forward-slash / or the backward-slash \ to separate the path elements.

It is safer to use an escape sequence when using the backward slash.

```
>>> os.chdir('C:\\Python33')
>>> print(os.getcwd())
C:\Python33
```

--------List Directories and Files

All files and sub-directories inside a directory can be retrieved using the listdir() method.

This method takes in a path and returns a list of subdirectories and files in that path. If no path is specified, it returns the list of subdirectories and files from the current working directory.

```
>>> print(os.getcwd())
C:\Python33
>>> os.listdir()
['DLLs','Doc','include','Lib','libs','LICENSE.txt','NEWS.txt','python.exe','pythonw.exe','README.txt','Scripts','tcl','Tools']
>>> os.listdir('G:\\')
['$RECYCLE.BIN','Movies','Music','Photos','Series','System Volume Information']
```

----------Making a New Directory

We can make a new directory using the mkdir() method.

This method takes in the path of the new directory. If the full path is not specified, the new directory is created in the current working directory.

```
>>> os.mkdir('test')
>>> os.listdir()
['test']
```

----------Renaming a Directory or a File

The rename() method can rename a directory or a file.

For renaming any directory or file, the rename() method takes in two basic arguments: the old name as the first argument and the new name as the second argument.

```
>>> os.listdir()
['test']
>>> os.rename('test','new_one')
>>> os.listdir()
['new_one']
```

----------Removing Directory or File

A file can be removed (deleted) using the remove() method.

Similarly, the rmdir() method removes an empty directory.

```
>>> os.listdir()
```

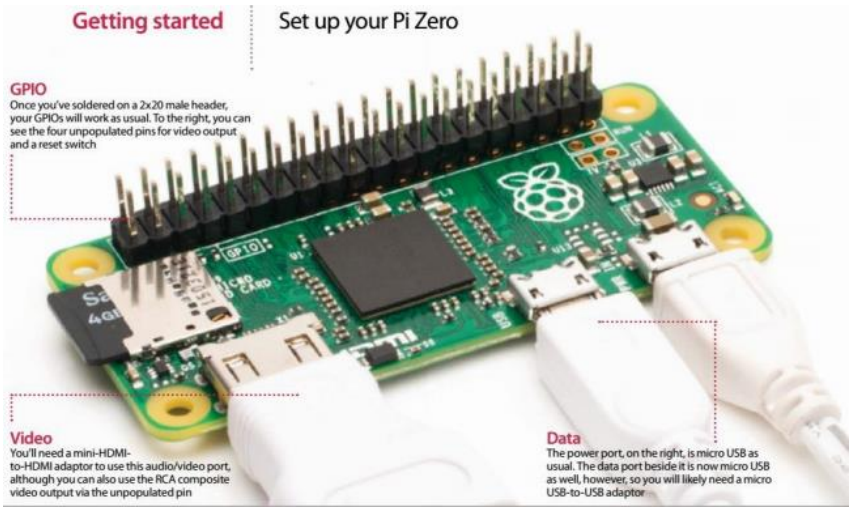
```
['new_one', 'old.txt']

>>> os.remove('old.txt')>>> os.listdir()

['new_one']

>>> os.rmdir('new_one')>>> os.listdir()

[]
```

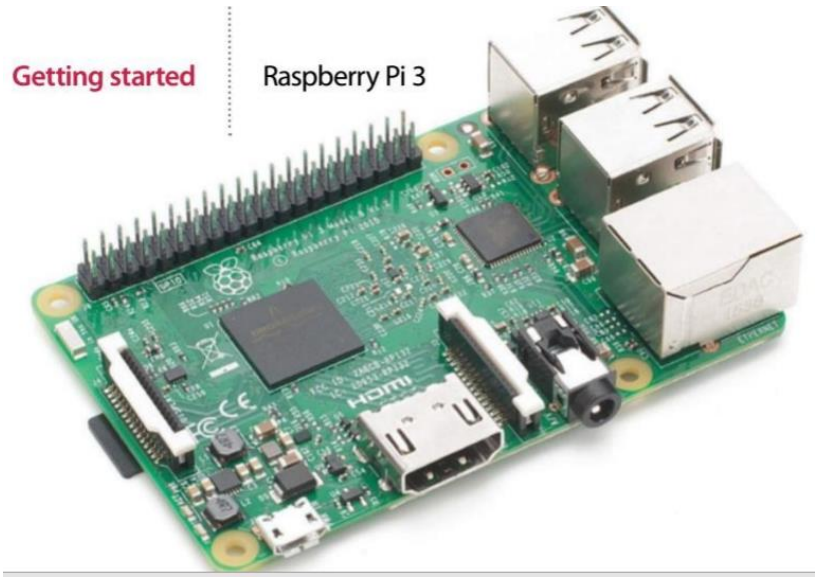
Note: The `rmdir()` method can only remove empty directories. In order to remove a non-empty directory, we can use the `rmtree()` method inside the `shutil` module.

```
>>> os.listdir()

['test']

>>> os.rmdir('test')

Traceback (most recent call last):

...

OSError: [WinError 145] The directory is not empty: 'test'

>>> import shutil

>>> shutil.rmtree('test')>>> os.listdir()

[]
```

5.2 RASPBERRY PI PROGRAMMING

Raspberry Pi is a low-cost mini-computer with the physical size of a credit card. Raspberry Pi runs various flavors of Linux and can perform almost all tasks that a normal desktop computer can do. Raspberry Pi also allows interfacing sensors and actuators through the general purpose I/O pins.





5.2.0 Raspberry Pi Interfaces

1. **Serial:** The serial interface on Raspberry Pi has receive (Rx) and transmit (Tx) pins for communication with serial peripherals.

2. **SPI:** Serial Peripheral Interface (SPI) is a synchronous serial data protocol used for communicating with one or more peripheral devices.
3. **I2C:** The I2C interface pins on Raspberry Pi allow you to connect hardware modules. I2C interface allows synchronous data transfer with just two pins - SDA (data line) and SCL (clock line).

LAB SESSION

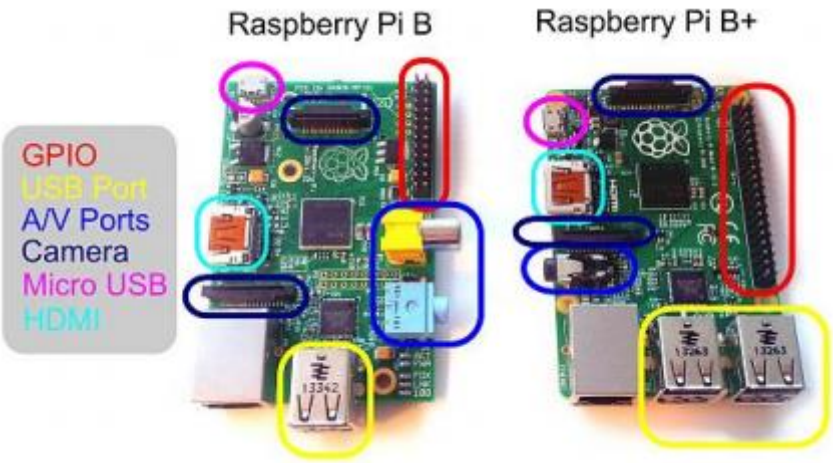
5.2.1 Raspbian Operating System

Raspbian is a Debian-based computer operating system for Raspberry Pi. Since 2015 till now it is officially provided by the Raspberry Pi Foundation as the primary operating system for the family of Raspberry Pi single-board computers. • Raspbian was created by Mike Thompson and Peter Green as an independent project. • The initial build was completed in June 2012. • The operating system is still under active development. Raspbian is highly optimized for the Raspberry Pi line's low-performance ARM CPUs.

Raspbian is the name given to a customized variant of the popular Debian Linux distribution. Debian is one of the longest-running Linux distributions and concentrates on stability, high compatibility, and excellent performance even on modest hardware. Raspbian takes Debian as its base, or parent distribution, and adds custom tools and software to make using the Raspberry Pi as easy as possible. To use Raspbian, you must enter a username and password. The default username is pi, and the default password is raspberry;

- **Installing Raspbian Operating System on Raspberry Pi.**

Required Components



a) **SD Card:** We recommend an 8GB class 4 SD card.

b) **Display and connecting cables:** Any HDMI/DVI monitor or TV should work as a display for the Pi. For best results, use one with HDMI input, but other connections are available for older devices.

c) **Keyboard and mouse:** Any standard USB keyboard and mouse will work with your Raspberry Pi.

d) **Power supply**: Use a 5V micro USB power supply to power your Raspberry Pi. Be careful that whatever power supply you use outputs at least 5V; insufficient power will cause your Pi to behave unexpectedly.

e) **Internet connection**: To update or download software, we recommend that you connect your Raspberry Pi to the internet either via an Ethernet cable or a WiFi adaptor.

- **Installing Raspbian Debian Wheezy Operating System Using Windows**

Step1. Download the file "RASPBIAN Debian Wheezy.zip" and extract the image file. (http://downloads.raspberrypi.org/raspbian_latest)

Step2. Insert the SD card into your SD card reader (format the sd card) and check which drive letter was assigned.

Step3. Download the **Win32DiskImager** utility from the Sourceforge Project page (it is also a zip file); you can run this from a USB drive.
(http://sourceforge.net/projects/win32diskimager/files/latest/download)

Step4. Extract the executable from the zip file and run the Win32DiskImager utility; you may need to run the utility as administrator.

Step5. Select the image file you extracted above.

Step6. Select the drive letter of the SD card in the device box. Be careful to select the correct drive; if you get the wrong one you can destroy your data on the computer's hard disk!

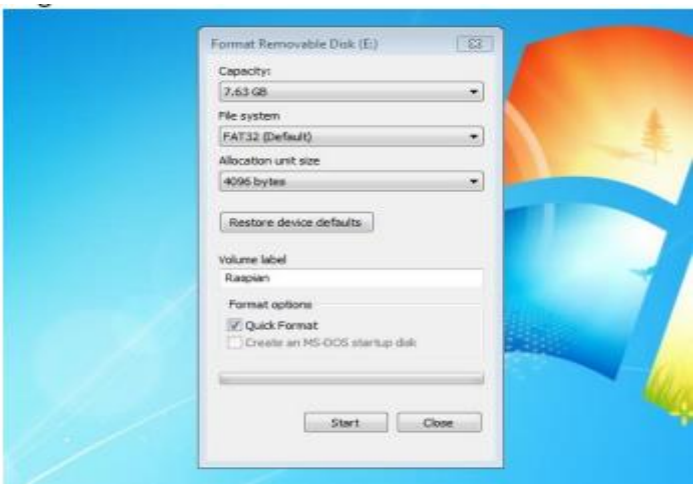
Step7. Click Write and wait for the write to complete.

Step8. Exit the imager and eject the SD card

1) FORMAT THE SD CARD

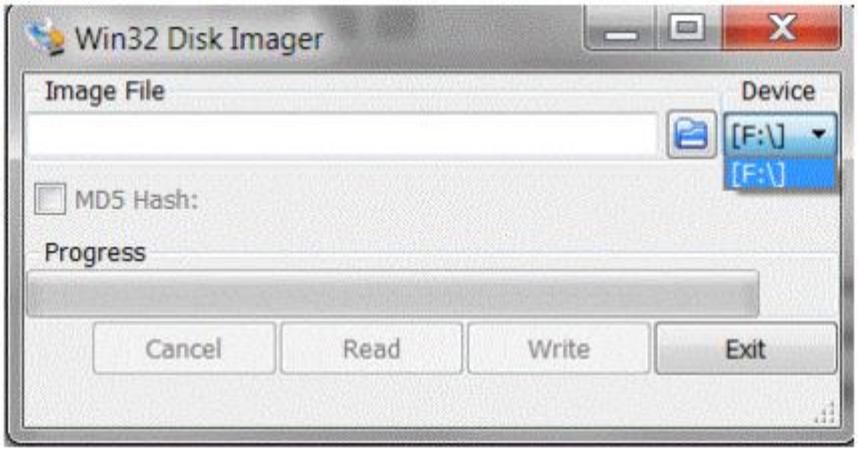
Locate your SD card drive, in Windows Explorer, and secondary-click the mouse to bring up the context-senstive menu. From the menu select Format….

Ensure that the option FAT32 (Default) is selected and click Start.

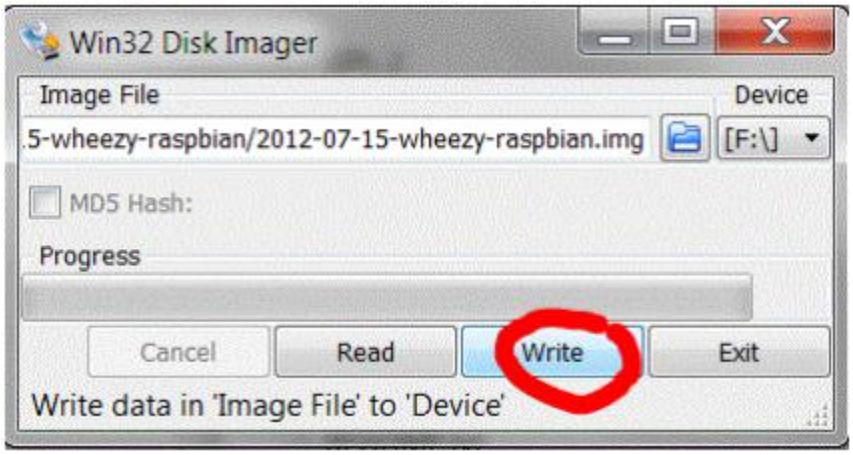


2) USING WIN32DISKIMAGER

choose the drive with your SD card to write the OS image on Then click on the folder icon and choose the unzipped .img file from earlier that you want to put on the SD card. Then click Write, to write the Operating system on the card from the .img file.

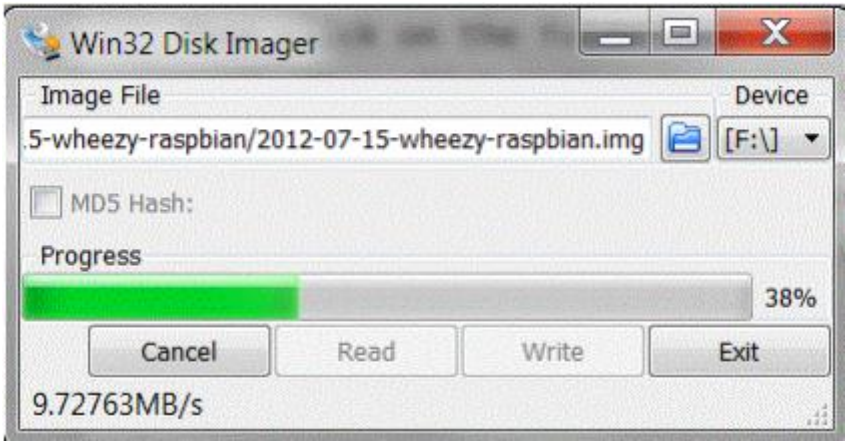


Write OS image from .img file to SD card

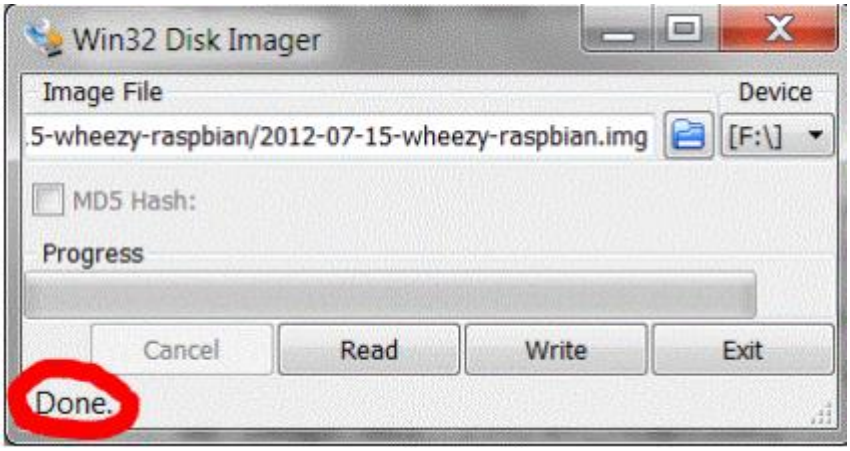


Check device and confirm

Progress indicator



Finished



Then you can eject the card reader and remove the SD card. Then you can try it out in your Raspberry Pi

5.2.2 PLUGGING IN YOUR RASPBERRY PI

a) Begin by slotting your SD card into the SD card slot on the Raspberry Pi.

b) Next, plug in your USB keyboard and mouse into the USB slots on the Raspberry Pi. Make sure that your monitor or TV is turned on

c) Then connect your HDMI cable from your Raspberry Pi to your monitor or TV

d) If you intend to connect your Raspberry Pi to the internet, plug in an Ethernet cable into the Ethernet port next to the USB ports

e) Plug in the micro USB power supply. This action will turn on and boot your Raspberry Pi.

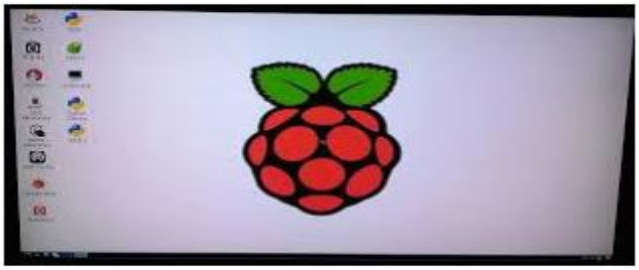
f) If this is the first time your Raspberry Pi SD card has been used, then you will have to select an operating system and configure it.

5.2.3 LOGGING INTO YOUR RASPBERRY PI

a) Once your Raspberry Pi has completed the boot process, a login prompt will appear. The default login for Raspbian is username pi with the password raspberry. Note you will not see any writing appear when you type the password. This is a security feature in Linux.

b) After you have successfully logged in, you will see the command line prompt **pi@raspberrypi~$**

c) To load the graphical user interface, type startx and press Enter on your keyboard



- **DOWNLOAD AND INSTALL WIRING PI:**

ONLINE INSTALL

If you do not have GIT installed, then under any of the Debian releases (e.g. Raspbian), you can install it with:

sudo apt-get update

sudo apt-get upgrade

To obtain WiringPi using GIT:

git clone git://git.drogon.net/wiringPi

If you have already used the clone operation for the first time, then

cd wiringPi

git pull origin

To build/install there is a new simplified script:

cd wiringPi

./build

The new build script will compile and install it all for you

- **TEST WIRINGPI'S INSTALLATION**

run the gpio command to check the installation:

gpio -v

gpio readall

That should give you some confidence that it's working OK

5.2.4 Creating a New User Account

Linux is at heart a social operating system designed to accommodate numerous users. By default, Raspbian is configured with two user accounts: pi, which is the normal user account, and root, which is a superuser account with additional permissions.

While it's certainly possible for you to use the pi account, you may prefer to create your own dedicated user account. Further accounts can also be created for friends or family members who might want to use the Pi.

1) Creating a new account on the Pi is straightforward and is roughly the same on all distributions, except for the username and password used to log in to the Pi initially. Just follow these steps:
2) Log in to the Pi using the existing user account (username pi and password raspberry if you're using the recommended Raspbian distribution). 2. Type the following as a single line with no spaces after any of the commas:

```
sudo useradd -m -G
  adm,dialout,cdrom,sudo,audio,video,plugdev,games,users,↩
input,netdev,gpio,i2c,spi username
```

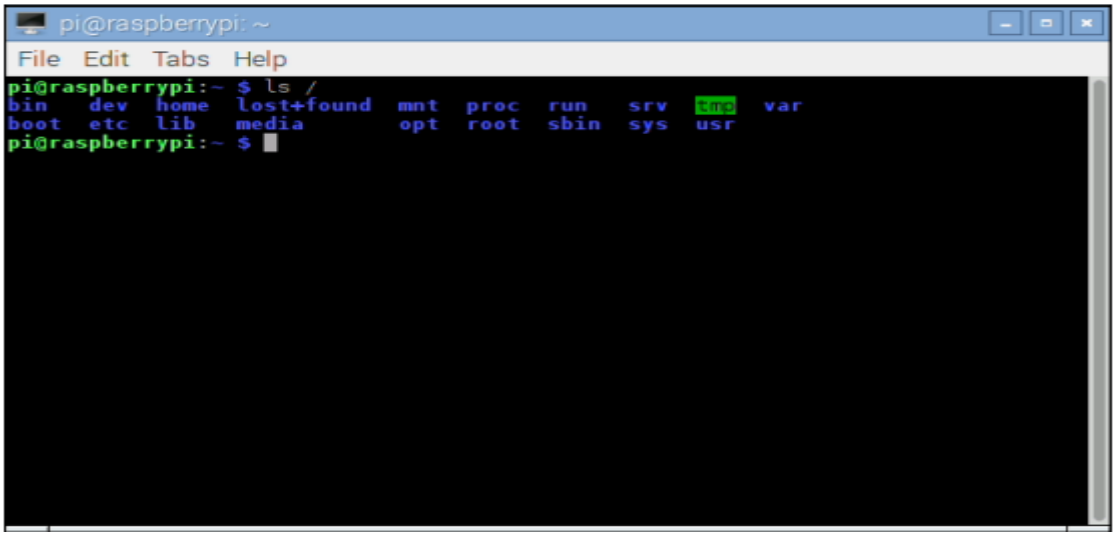
   This creates a new, empty user account.
3) To set a password on the new account, type **sudo passwd username** followed by the new password when prompted.

The **useradd** command says you want to create a new user account. The **-m** section tells the useradd program to create a home directory where the new user can store his or her files. The big list following the -G flag is the list of groups of which the user should be a member.

5.2.5 File System Layout

The content of the SD card is known as its file system and is split into multiple sections, each with a particular purpose.

If you log in to the Pi and type ls /, you'll see various directories displayed (see Figure 3-3). Some of these are areas of the SD card for storing files, while others are virtual directories for accessing different portions of the operating system or hardware.



The directories visible on the default Raspbian distribution are as follows:

- **boot**—This contains the Linux kernel and other packages needed to start the Pi.
- **bin**—Operating system-related binary files, such as those required to run the GUI, are stored here.
- **dev**—This is a virtual directory, which doesn't actually exist on the SD card. All the devices connected to the system—including storage devices and the sound card—can be accessed from here.
- **etc**—This stores miscellaneous configuration files, including the list of users and their encrypted passwords.
- **home**—All users get a subdirectory beneath this directory to store all their personal files.
- **lib**—This is a storage space for libraries, which are shared bits of code required by numerous different applications.
- **lost+found**—This is a special directory where file fragments are stored if the system crashes.
- **media**—This is a special directory for removable storage devices, such as USB memory sticks or external CD drives.
- **mnt**—This folder is used to manually mount storage devices, such as external hard drives.
- **opt**—This stores optional software that is not part of the operating system. If you install new software to your Pi, it will usually go here or in usr.
- **proc**—This is another virtual directory, containing information about running programs, which are known in Linux as processes.

- **run**—A special directory used by various daemons, processes which run in the background.
- **root**—Although most user's files are kept in a subdirectory of home, the root (superuser) account's files are kept here.
- **sbin**—This stores special binary files, primarily used by the root (superuser) account for system maintenance.
- **srv**—A directory used to store data served by the operating system. This is empty in Raspbian.
- **sys**—This is a virtual directory storing system information used by the Linux kernel.
- **tmp**—Temporary files are stored here automatically.
- **usr**—This directory provides storage for user-accessible programs.
- **var**—This is a directory which programs use to store changing values or variables.

- **Finding the Software You Want**

The first step to installing a new piece of software is to find out what it's called. The easiest way to do so is to search the **cache** of available software packages. This cache is a list of all the software available to install via **apt**, stored on Internet servers known as repositories.

The apt software includes a utility for managing this cache, called **apt-cache**. Using this software, it's possible to run a search on all the available software packages for a particular word or phrase. For example, to find a game to play, you can type the following command:

- **apt-cache search game**

5.2.6 Installing Software

Once you know the name of the package you want to install, switch to the apt-get command to install it. Installing software is a privilege afforded only to the root user, as it affects all users of the Raspberry Pi. As a result, the commands will need to be prefaced with sudo to tell the operating system that it should be run as the root user.

For example, to install the package **nethack-console** :

- **sudo apt-get install nethack-console**

5.2.7 Uninstalling Software

If you decide you no longer want a piece of software, apt-get also includes a remove command that cleanly uninstalls the package along with any dependencies that are no longer required. When you're using a smaller SD card with the Pi, your ability to try out software and quickly remove it is very useful.

To remove nethack-console, simply open the terminal and type the following command:

- **sudo apt-get remove nethack-console**

The remove command has a more powerful brother in the form of the purge command. Like remove, the **purge** command gets rid of software you no longer require. Where remove leaves the software's configuration files intact, however, purge removes everything.

- **sudo apt-get purge nethack-console**

5.2.8 Upgrading Software

In addition to installing and uninstalling packages, apt can be used to keep them up-to-date. Upgrading a package through apt ensures that you've received the latest updates, bug fixes, and security patches.

- **sudo apt-get update**

When upgrading software, you have two choices: you can upgrade everything on the system simultaneously or upgrade individual programs. If you just want to keep your entire distribution updated, the former is achieved by typing the following:

- **sudo apt-get upgrade**

To upgrade an individual package, simply tell apt to install it again. For example, to install a nethack-console upgrade, you type this:

- **sudo apt-get install nethack-console**

Shutting the Pi Down Safely

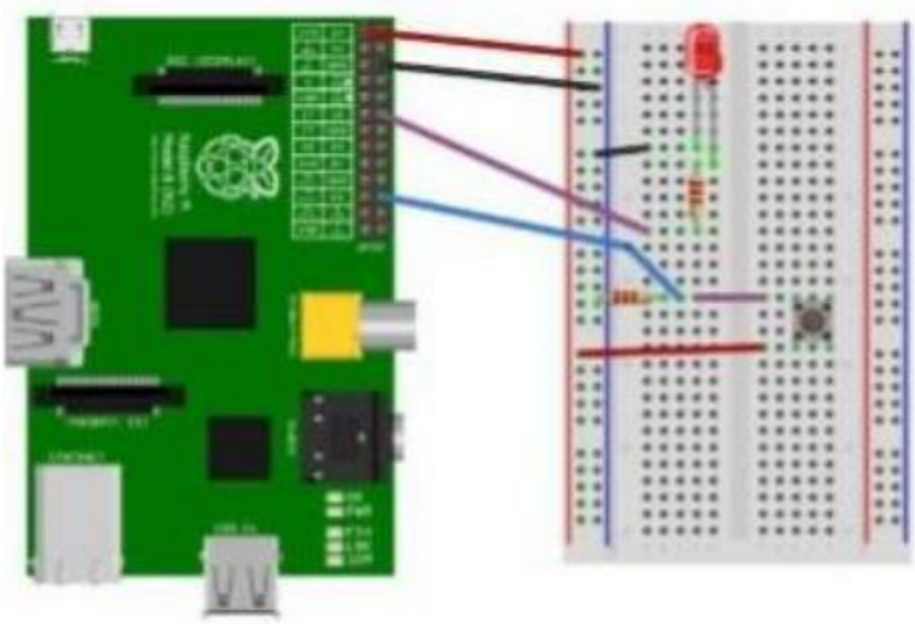
Although the Pi doesn't have a power switch like a traditional computer, that doesn't mean you can simply pull the power cable out when you're finished. Even when it doesn't appear to be doing anything, the Pi is often reading from or writing to the SD card, and if it loses power while this is happening, the contents of the card can become corrupt and unreadable.

To shut down using the terminal, type the following command:

- **sudo shutdown -h now**

5.2.9 Raspberry Pi Example: Interfacing LED and switch with Raspberry Pi

- **Circuit Diagram**



- **Program Code**

```
from time import sleep
 import RPi.GPIO as GPIO GPIO.setmode(GPIO.BCM)
#Switch Pin GPIO.setup(25,GPIO.IN)
 #LEDPin
GPIO.setup(18,GPIO.OUT)
state=false
def toggleLED(pin):
    state = not state
    GPIO.output(pin,state)
whileTrue:
        try:
            if (GPIO.input(25) ==True):
```

```
            toggleLED(pin)
            sleep(.01)
    except KeyboardInterrupt:
        exit()
```

REFERENCES

1. Postscapes.com. (2017). IoT Standards & Protocols Guide | 2018 Comparisons on Network, Wireless Comms, Security, Industrial. [online] Available at: https://www.postscapes.com/internet-of-things-protocols
2. Sundmaeker, H. (2010). Vision and challenges for realising the Internet of Things. Luxembourg: Publications Office of the European Union.
3. Shi-Wan, L., Miller, B. and Durand, J. (2017). The Industrial Internet of Things Volume G1: Reference Architecture. 1st ed. [ebook] Industrial Internet Consortium. Available at: https://www.iiconsortium.org/IIC_PUB_G1_V1.80_2017-01-31.pdf
4. Carrez, F. (2018). Open Platforms – IoT-A Architectural Reference Model. [online] Open-platforms.eu. Available at: http://open-platforms.eu/standard_protocol/iot-a-architectural-referencemodel
5. Walter, J. (2017). The essential building blocks of an IoT architecture. [online] Mobile Business Insights. Available at: https://mobilebusinessinsights.com/2017/09/the-essential-buildingblocks-of-an-iot-architecture/