

Introduction to Python

C vs Python

C

- Compiled before executing
- Variables & types need to be declared
- Not object-oriented
- Limited built-in functions
- Simple data structures
- Relatively hard syntax

- *Fast*
- *Complicated*

Python

- Interpreted
- No type declaration required
- Object-oriented
- Large number of built-in functions
- Complex data structures
- Very easy and intuitive syntax

- *Slow*
- *Easy*

Why do we need Python?

Answer: **Rapid Prototyping**

- Codes in C/C++/Java have a very large size
 - Every small detail regarding variable types, segment sizes, function parameters, return types, etc. need to be given attention.
 - Very low flexibility.
 - Debugging is very difficult.
-
- **Python** makes prototyping easier due to its **flexible** and **intuitive** nature.

Starting up...

Running the interpreter:

Linux: `$python3`

Running a script:

Linux: `$python3 filename.py`

The usual Hello World

// Hello world in C:

```
#include <stdio.h>
```

```
int main()
```

```
{
```

```
    printf("Hello World! \n");
```

```
    return 0;
```

```
}
```

#Hello World in Python:

```
print "Hello World!"
```

Basics:

1. Variables & data types
2. Lists and Dictionaries
3. Conditions
4. Loops
5. Functions
6. Classes & Objects
7. Libraries

Variables and Constants

#Declaring variables in Python:

```
var1 = 4  
var2 = 3.14  
var3 = "ERC"  
var4 = True
```

#Printing a variable

```
print (var3)
```

Type flexibility in python

#Same variable, different types

```
X = 10
type(X)
X = False
type(X)
X = "Rishikesh"
type(X)

X = (X != "Rishikesh")
print(X)
type(X)
```

#Common operations for different types

```
X = "BITS"
X = X + "Goa"
print(X)
print(5*X)

Y = True
print(Y + 41)

print( (True + 3)* "F" )
```


Lists

#Declaring a list

```
list1 = [ ]
```

```
list2 = [3, 4, 5, 6, 7]
```

#Accessing elements:

```
list2[0]          #0th element
```

```
list2[2:4]        #From 2nd(inclusive) to 4th(exclusive)
```

```
list2[:3]         #Upto 3rd element (exclusive)
```

```
List2[3:]         #From 3rd element (inclusive) onwards
```

```
List2[-2]         #Second-last element
```

Lists - basic operations

#Length of list:

```
len(list2)
```

#adding elements to a list:

```
list2.append(2019)
```

```
list2.insert(3, "ABCD")
```

```
list2 = list2 + [True, 412, 'a', 'b']
```

#Insert 2019 at the end

#insert "ABCD" at position 3

#Concatenate

#Printing a list:

```
list2
```

```
print(list2)
```

More operations

Try out the following functions on lists:

```
List = [4, 5, 3, 2, 13, 1, 7, 6]
```

```
list.count(3)
```

```
list.append(3)
```

```
list.count(3)
```

```
list.index(13)
```

```
sum(list)
```

```
min(list)
```

```
max(list)
```

```
list.sort()
```

?

What would the
corresponding C code for
these functions look like?

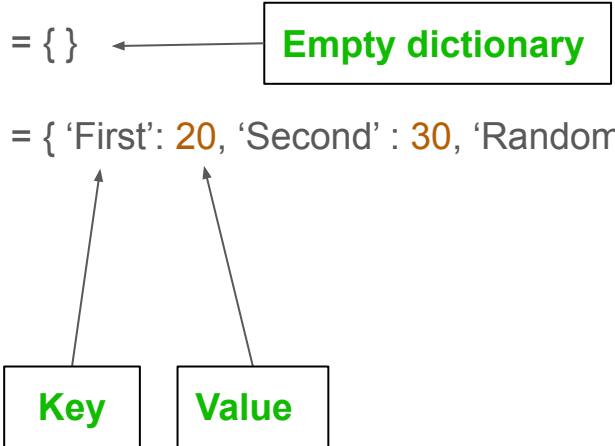
Dictionaries

#Creating a dictionary:

dict1 = { } ← **Empty dictionary**

dict2 = { 'First': 20, 'Second': 30, 'Random': [5, 6, 7] }

Key **Value**



Accessing elements:

```
dict2['Random']  
dict2['Second']
```

#Adding elements

```
dict2['ABC'] = 5
```

#Printing a dictionary

```
print(dict2)
```

Conditions:

SYNTAX:

```
if <condition> :  
    <statement1>  
    <statement2>  
    ...  
    <statement n>  
else:  
    <statement1>  
    <statement2>  
    ...  
    <statement n'>
```

- No brackets required
- **Do not forget indentation**

EXAMPLE:

```
a = 5  
  
if a == 4 :  
    print("a is 4")  
else:  
    print("a is not 4")
```

Multiple conditions & boolean operations

Conditions can be coupled together using boolean operators as follows:

if <i><condition1></i> and <i><condition2></i> :	#True only if both conditions are True
if <i><condition1></i> or <i><condition2></i> :	#True if one or both conditions are True
if not <i><condition></i> :	#True only if the condition is False

'in' operator

The 'in' operator in Python provides an easy way to have conditional statements involving lists/dictionaries.

SYNTAX:

<value> in <list>

#checks if the given value is in the given list

#Example:

```
list1 = [3, 4, 5, 6]
```

```
print( 3 in list1)
```

```
print( 10 in list1)
```

For-Loops

Method 1: Iterating over indices

```
list = [1, 2, 'A', False]
```

```
for i in range(4):  
    print( list[i] )
```

- **Automatic iteration** (i++ not required)
- **range**(n) denotes i goes from 0 to n.
- **Indentation necessary**

Method 2: Iterating over list elements directly

```
list = [1, 2, 'A', False]
```

```
for i in list:  
    print( i )
```

- **Automatic iteration** over list **elements**
- Here, **i** is not the index of the element, but the **element itself**
- **Indentation necessary**

While loops

- Similar to for loops, except iteration is not automatic
- **Indentation necessary**
- SYNTAX:

```
while <condition> :  
    <statement1>  
    <statement2>  
    ...  
    <statement n>
```

Exercise -1:

A bot has 3 behaviours: “move”, “jump” and “stop”

A straight path is given in the form of a list. Elements of the list denote the following:

True: Free path (*the robot can **move***)

False: Obstacle (*the robot needs to **jump***)

“Goal”: Destination (*the robot has to **stop***)

Problem: Make a list of commands that need to be given to the bot sequentially and print the list. Also count the number of jumps needed.

Example list: [True, True, False, True, False, “Goal”]

Output list: [“move”, “move”, “jump”, “move”, “jump”, “stop”,]

Number of jumps: 2

**(Try solving the problem once using for loops and once using while loops separately)*

Functions

The **def** keyword is used to define functions

Function definitions in C:

1. Return type
2. Name
3. Parameter list *with types*

Function definitions in Python:

1. Name
2. Parameter list

No data types need to be mentioned
Indentation necessary

SYNTAX:

Function Definition -

```
def function_name(arg1, arg2, arg3):  
    <statement1>  
    <statement2>  
    ...  
    <statement n>  
    return ret1, ret2, ret3
```

Function call -

```
val1, val2, val3 = function_name(par1, par2, par3)
```

Default parameter values & parameter ordering

EXAMPLE:

Definition-

```
def func1(var1, var2, show_status=False):  
    diff = var1 - var2  
    sum = var1 + var2  
    if(show_status):  
        print("Success!")  
    return diff, sum
```

Calling-

#1

```
a, b = func1(3, 4)
```

#2

```
a, b = func1(3, 4, True)
```

#3

```
a, b = func1(var2 = 4, show_status=True, var1 = 3)
```

Libraries:

Collection of predefined functions and classes for specific purposes.

Allows the user to directly use optimized code for advanced & complex tasks.

Commonly used Python libraries:

- numpy (*for matrix-operations*)
- matplotlib (*for plotting graphs*)
- scipy (*for scientific calculations*)
- opencv (*for image processing*)

Commonly used libraries in robotics

- RPi.GPIO (*for using Raspberry Pi pins*)
- serial (*For serial communication*)
- rospy (*for using ROS*)

Pyserial

Arduino IDE - `Serial.println(data);`



1. Where is **data** being printed?
2. Who is sending this data?
3. What is a COM port?

Hardware-Software Interface:

- Data from the microcontroller is sent to the computer **serially** through a '**port**'.
- **Pyserial** allows us to access data coming from the port.
- This data can now be processed using python scripts.
- The processed output can then be sent back to the microcontroller

Pyserial - Basics

Installing:

```
$pip3 install pyserial
```

Importing:

```
import serial
```

Initializing

```
ser = serial.Serial('COM3', baudrate=9600, timeout = 1)
```

Pyserial basics (continued)

```
ser = serial.Serial( 'COM3', baudrate = 9600, timeout = 1 )
```

Port Name-

Needs to match the port used for uploading from the IDE

Baud Rate-

Needs to match the baud rate used in `Serial.begin()`;

Timeout-

No. of seconds to wait if no data is being received

Pyserial basics (continued)

Reading the data:

```
d = ser.readline()
```

#The latest line of data received is now stored in the variable **d**.

#This variable can now be used as any other python variable.

```
print(d)
```

```
print(type(d))
```

```
print(d.decode('ascii'))
```

#What is the difference in the outputs of the 2 prints? Why?

Pyserial basics (continued)

Sending data from computer to microcontroller:

Python script - `ser.write(var)`

#Sends the value of **var** on the serial port

Arduino IDE - `Serial.read();`

//The arduino reads the data received from the COM port

Example code

On Arduino IDE:

```
void setup() {  
    Serial.begin(9600);  
}  
  
void loop() {  
    char c;  
    if(Serial.available()){  
        c = Serial.read();  
    }  
    Serial.println(c);  
    delay(10);  
}
```

On python:

```
from time import sleep  
import serial  
  
ser = serial.Serial('COM3', 9600)  
  
num=64  
while num <= 90 :  
    ser.write(str(num))  
    sleep(0.5)
```

Home - Exercise

Write a **Python code** to detect the approximate amount of time (in seconds) for which an IR sensor is sees an obstacle.

Whenever a new obstacle is seen, print “New Obstacle”

Whenever the obstacle disappears, print the amount of time for which it was present.

NOTE: Use a physical IR sensor with ESP32. You can use any arbitrary object as the obstacle.

Documentation of Python Libraries:

Using libraries extensively becomes necessary while developing in Python.

It is not possible to remember all functions, parameters, return types, etc. from these libraries, even if they are frequently used.

Hence, it becomes necessary to frequently visit the **official documentation** of these libraries.

Links for some of them have been given below:

- numpy → <https://numpy.org/>
- opencv → <https://opencv.org/>
- RPi.GPIO → <https://www.raspberrypi.org/documentation/usage/gpio/python/README.md>
- serial → <https://pyserial.readthedocs.io/en/latest/shortintro.html>

Object Oriented Programming in Python

Predefined types for variables: int, str, bool, float, etc.

*Each type has some unique behaviour

What if I want a variable **p1** to be of type '**animal**'?

Class: A *blueprint* for a complex user-defined “type”.

Object: A variable of the type defined by the class. (a.k.a. “**Instance**” of a class)

Classes:

Properties of an animal

Features of an animal:

1. Size
2. Colour
3. Number of legs



Class: Animal

Attributes (internal variables) :

1. size
2. colour
3. num_legs

What can an animal do?

1. Eat
2. Sleep
3. Run



Methods (functions):

1. *eat*(food_type)
2. *sleep*(duration)
3. *run*(duration, speed)

Class Structure:

Class Definition:

- **Constructor**
- **Attributes**
- **Methods**