

## Python Introduction

- Python is a general purpose, dynamic, high level and interpreted programming language.
- It supports Object Oriented programming approach to develop applications.
- It is simple and easy to learn and provides lots of high-level data structures.
- Python is easy to learn yet powerful and versatile scripting language which makes it attractive for Application Development.
- Python's syntax and dynamic typing with its interpreted nature, makes it an ideal language for scripting and rapid application development.
- Python supports multiple programming pattern, including object oriented, imperative and functional or procedural programming styles.
- Python is not intended to work on special area such as web programming. That is why it is known as multipurpose because it can be used with web, enterprise, 3D CAD etc.
- We don't need to use data types to declare variable because it is dynamically typed so we can write `a=10` to assign an integer value in an integer variable.
- Python makes the development and debugging fast because there is no compilation step included in python development and edit-test-debug cycle is very fast.

## Python History

- Python laid its foundation in the late 1980s.
- The implementation of Python was started in the December 1989 by Guido Van Rossum at CWI (Centrum Wiskunde & Informatica (CWI) is the national research institute for mathematics and computer science in the Netherlands) in Netherlands.
- In February 1991, van Rossum published the code (labeled version 0.9.0) to alt.sources.
- On December 3, 2008, Python 3.0 (also called "Py3K") was released.
- Python is influenced by following programming languages:
  - ABC language.
  - Modula-3

# What is Python ?

- Python is a simple, easy to learn, powerful, high level and object-oriented programming language.
- Python is an interpreted scripting language also.
- Guido Van Rossum is known as the founder of python programming

## Python Features

### 1) Easy to Learn and Use

- Python is easy to learn and use.
- It is developer-friendly and high level programming language.

### 2) Expressive Language

- Python language is more expressive means that it is more understandable and readable.

### 3) Interpreted Language

- Python is an interpreted language
- i.e. interpreter executes the code line by line at a time.
- This makes debugging easy and thus suitable for beginners.

### 4) Cross-platform Language

- Python can run equally on different platforms such as Windows, Linux, Unix and Macintosh etc.
- we can say that Python is a portable language.

### 5) Free and Open Source

- Python language is freely available at official web address.
- The source-code is also available. Therefore it is open source.

### 6) Object-Oriented Language

- Python supports object oriented language and concepts of classes and objects come into existence.

### 7) Extensible

- It suggests that other languages such as C/C++ can be used to compile the code and as a result it can be used further in our python code.

## 8) Large Standard Library

- Python has a large and broad library and provides rich set of module and functions for rapid application development.

## 9) GUI Programming Support

- Graphical user interfaces can be developed using Python.

## 10) Integrated

- It can be easily integrated with languages like C, C++, JAVA etc.

## Python Version

- Python programming language is being updated regularly with new features and supports.
- There are lots of updations in python versions, started from 1994 to current release.

Python Version	Release Date
Python 1.0	January 1994
Python 1.5	December 31, 1997
...	...
Python 3.3	September 29, 2012
Python 3.4	March 16, 2014
...	....
Python 3.6.4	December 19, 2017

## Python Applications Area

- Python is known for its general purpose nature that makes it applicable in almost each domain of software development.
- Python as a whole can be used in any sphere of development.

- 1) Web Applications
- 2) Desktop GUI Applications
- 3) Software Development
- 4) Scientific and Numeric
- 5) Business Applications
- 6) Console Based Application
- 7) Audio or Video based Applications
- 8) 3D CAD Applications
- 9) Enterprise Applications
- 10) Applications for Images

## Python Example

- Python is easy to learn and code and can be execute with python interpreter.
- We can also use Python interactive shell to test python code immediately.

- A simple hello world example is given below.
- Write below code in a file and save with **.py** extension.
- Python source file has .py extension

```
print("hello world by python!")
```

- Execute this example by using following command.

```
hello.py
```

- **Output**

```
hello world by python!
```

## **Python Example using Interactive Shell**

- Python interactive shell is used to test the code immediately and does not require to write and save code in file.
- Python code is simple and easy to run.
- Here is a simple Python code that will print "Welcome to Python".
- A simple python example is given below.

```
a="Welcome To Python"  
print a
```

```
Welcome To Python
```

- In python 3.4 version, you need to **add parenthesis ()** in a string code to print it.

```
a=("Welcome To Python Example")  
print a      (Before Python 3.4)  
print (a)    (in Python 3.4)
```

```
Welcome To Python Example
```

## **Python Variables**

- Variable is a name which is used to refer memory location.
- Variable also known as identifier and used to hold value.

- In Python, we don't need to specify the type of variable because Python is a type infer language and smart enough to get variable type.
- Variable names can be a group of both letters and digits, but they have to begin with a letter or an underscore.
- It is recommended to use lowercase letters for variable name.
- Demo and demo both are two different variables.

**Note - Variable name should not be a keyword.**

- Python does not bound us to declare variable before using in the application.
- It allows us to create variable at required time.
- We don't need to declare explicitly variable in Python.
- When we assign any value to the variable that variable is declared automatically.
- The equal (=) operator is used to assign value to a variable.

## Multiple Assignment

- Python allows us to assign a value to multiple variables in a single statement which is also known as multiple assignment.
- We can apply multiple assignments in two ways either by assigning a single value to multiple variables or assigning multiple values to multiple variables.
- Lets see given examples.

- **Assigning single value to multiple variables**

```
x=y=z=50
```

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

- **Assigning multiple values to multiple variables**

```
a,b,c=5,10,15
print (a)
print (b)
print (c)
```

**Note:- The values will be assigned in the order in which variables appears**

# Python Data Types

- Variables can hold values of different data types.
- Python is a dynamically typed language hence we need not define the type of the variable while declaring it.
- The interpreter implicitly binds the value with its type.
- Python enables us to check the type of the variable used in the program.
- Python provides us the `type()` function which returns the type of the variable passed.
- Consider the following example to define the values of different data types and checking its type.

```
a=10  
b="Hi Python"  
c = 10.5
```

```
print(type(a));  
print(type(b));  
print(type(c));
```

## Standard data types

- A variable can hold different types of values.
- For example, a person's name must be stored as a string whereas its id must be stored as an integer.
- Python provides various standard data types that define the storage method on each of them.
- The data types defined in Python are given below.

- 1) Numbers
- 2) String
- 3) List
- 4) Tuple
- 5) Dictionary

### 1) Numbers

- Number stores numeric values.
- Python creates Number objects when a number is assigned to a variable.

- For example;  
`a = 3 , b = 5` #a and b are number objects
- Python supports 4 types of numeric data.
- int (signed integers like 10, 2, 29, etc.)
- long (long integers used for a higher range of values like 908090800L, -0x1929292L, etc.)
- float (float is used to store floating point numbers like 1.9, 9.902, 15.2, etc.)  
 complex (complex numbers like 2.14j, 2.0 + 2.3j, etc.)
- Python allows us to use a lower-case L to be used with long integers. However, we must always use an upper-case L to avoid confusion.
- A complex number contains an ordered pair, i.e.,  $x + iy$  where x and y denote the real and imaginary parts respectively).

## 2) String

- The string can be defined as the sequence of characters represented in the quotation marks.
- In python, we can use single, double, or triple quotes to define a string.
- String handling in python is a straightforward task since there are various inbuilt functions and operators provided.
- In the case of string handling, the operator + is used to concatenate two strings as the operation "hello"+" "
- python" returns "hello python".
- The operator \* is known as repetition operator as the operation "Python " \*2 returns "Python Python ".
- The following example illustrates the string handling in python.

```
str1 = 'hello Good Morning ' #string str1
str2 = ' how are you' #string str2
```

```
print (str1[0:2]) #printing first two character using slice operator
print (str1[4]) #printing 4th character of the string
print (str1*2) #printing the string twice
print (str1 + str2) #printing the concatenation of str1 and str2
```

### 3) List

- Lists are similar to arrays in C.
- The list can contain data of different types.
- The items stored in the list are **separated with a comma (,)** and enclosed **within square brackets []**.
- We can use slice [:] operators to access the data of the list.
- The concatenation operator (+) and repetition operator (\*) works with the list in the same way as they were working with the strings.
- Consider the following example.

```
i = [1, "hi", "How", 2]
print (i[3:])
print (i[0:2])
print (i)
print (i + 1)
print (1 * 3)
```

### 4) Tuple

- A tuple is similar to the list in many ways.
- tuples also contain the collection of the items of different data types.
- The items of the tuple are **separated with a comma (,)** and **enclosed in parentheses ()**.
- A tuple is a read-only data structure as we can't modify the size and value of the items of a tuple.
- Let's see a simple example of the tuple.

```
t = ("hi", "Hello", 2)
print (t[1:]);
print (t[0:1]);
print (t);
print (t + t);
print (t * 3);
print (type(t))
t[2] = "hi";
```



## 5) Dictionary

- Dictionary is an ordered set of a key-value pair of items.
- It is like an associative array or a hash table where each key stores a specific value.
- Key can hold any primitive data type whereas value is an arbitrary Python object.
- The items in the dictionary are **separated with the comma** and **enclosed in the curly braces {}**.

Consider the following example.

```
d = {1:'abc', 2:'xyz', 3:'pqr', 4:'abcd'}  
print("1st name is "+d[1])  
print("2nd name is "+ d[4])  
print (d)  
print (d.keys())  
print (d.values())
```

## Python Keywords

- Python Keywords are special reserved words which convey a special meaning to the compiler/interpreter.
- 
- Each keyword have a special meaning and a specific operation. These keywords can't be used as variable.
- Following is the List of Python Keywords

True	False	None	and	as
asset	def	class	continue	break
else	finally	elif	del	except
global	for	if	from	import
raise	try	or	return	pass
nonlocal	in	not	is	lambda

## Python Literals

- Literals can be defined as a data that is given in a variable or constant.
- Python support the following literals:

### 1) String literals:

- String literals can be formed by enclosing a text in the quotes.
- We can use both single as well as double quotes for a String.

Eg:

"abcd" , '12345'

## Types of Strings:

- There are two types of Strings supported in Python:

### a).Single line String-

- Strings that are terminated within a single line are known as Single line Strings.

Eg:

```
text1='hello'
```

### b).Multi line String-

- A piece of text that is spread along multiple lines is known as Multiple line String.
- There are two ways to create Multiline Strings

#### 1). Adding black slash at the end of each line.

```
text1='hello\ user'  
text1 'hellouser'
```

#### 2).Using triple quotation marks:-

```
str2="""welcome to SSSIT"""  
print str2
```

welcome to SSSIT

## 2) Numeric literals:

- Numeric Literals are unchallengeable.
- Numeric literals can belong to following four different numerical types.
  - Int(signed integers)
  - Long(long integers)
  - float(floating point)
  - Complex(complex)
- Numbers( can be both positive and negative) with no fractional part.  
eg: 100
- Integers of unlimited size followed by lowercase or uppercase L  
eg: 87032845L
- Real numbers with both integer and fractional part  
eg: -26.2

- In the form of  $a+bj$  where  $a$  forms the real part and  $b$  forms the imaginary part of complex number.  
eg:  $3.14j$

### 3) Boolean literals:

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

### 4) Special literals.

- Python contains one special literal i.e., None.
- None is used to specify to that field that is not created.
- It is also used for end of lists in Python.

```
val1=10  
val2=None  
print (val1)  
print (val2)
```

### 5) Literal Collections.

- Collections such as tuples, lists and Dictionary are used in Python.

## Python Operators

- The operator can be defined as a symbol which is responsible for a particular operation between two operands.
- Operators are the pillars of a program on which the logic is built in a particular programming language.
- Python provides a variety of operators described as follows
  - 1) Arithmetic operators
  - 2) Comparison operators
  - 3) Assignment Operators
  - 4) Logical Operators
  - 5) Bitwise Operators
  - 6) Membership Operators
  - 7) Identity Operators

### 1) Arithmetic operators

- Arithmetic operators are used to perform arithmetic operations between two operands.
- It includes  $+$ (addition),  $-$  (subtraction),  $*$ (multiplication),  $/$ (divide),  $\%$ (remainder),  $//$ (floor division), and exponent  $(**)$ .

- Consider the following table for a detailed explanation of arithmetic operators.

Operator	Description
<b>+</b> (Addition)	It is used to add two operands. For example, if $a = 20$ , $b = 10 \Rightarrow a + b = 30$
<b>-</b> (Subtraction)	It is used to subtract the second operand from the first operand. If the first operand is less than the second operand, the value result negative. For example, if $a = 20$ , $b = 10 \Rightarrow a - b = 10$
<b>/</b> (divide)	It returns the quotient after dividing the first operand by the second operand. For example, if $a = 20$ , $b = 10 \Rightarrow a / b = 2$
<b>*</b> (Multiplication)	It is used to multiply one operand with the other. For example, if $a = 20$ , $b = 10 \Rightarrow a * b = 200$
<b>%</b> (reminder)	It returns the reminder after dividing the first operand by the second operand. For example, if $a = 20$ , $b = 10 \Rightarrow a \% b = 0$
<b>**</b> (Exponent)	It is an exponent operator represented as it calculates the first operand power to second operand.
<b>//</b> (Floor division)	It gives the floor value of the quotient produced by dividing the two operands.

## 2) Comparison operators

- Comparison operators are used to comparing the value of the two operands and returns boolean true or false accordingly.
- The comparison operators are described in the following table.

Operator	Description
<b>==</b>	If the value of two operands is equal, then the condition becomes true.
<b>!=</b>	If the value of two operands is not equal then the condition becomes true.
<b>&lt;=</b>	If the first operand is less than or equal to the second operand, then the condition becomes true.
<b>&gt;=</b>	If the first operand is greater than or equal to the second operand, then the condition becomes true.
<b>&lt;&gt;</b>	If the value of two operands is not equal, then the condition becomes true.
<b>&gt;</b>	If the first operand is greater than the second operand, then the condition becomes true.
<b>&lt;</b>	If the first operand is less than the second operand, then the condition becomes true.

## 3) Assignment Operators

- The assignment operators are used to assign the value of the right expression to the left operand.
- The assignment operators are described in the following table.

Operator	Description
<b>=</b>	It assigns the the value of the right expression to the left operand.

+=	It increases the value of the left operand by the value of the right operand and assign the modified value back to left operand. For example, if $a = 10$ , $b = 20 \Rightarrow a + = b$ will be equal to $a = a + b$ and therefore, $a = 30$ .
-=	It decreases the value of the left operand by the value of the right operand and assign the modified value back to left operand. For example, if $a = 20$ , $b = 10 \Rightarrow a - = b$ will be equal to $a = a - b$ and therefore, $a = 10$ .
*=	It multiplies the value of the left operand by the value of the right operand and assign the modified value back to left operand. For example, if $a = 10$ , $b = 20 \Rightarrow a * = b$ will be equal to $a = a * b$ and therefore, $a = 200$ .
%=	It divides the value of the left operand by the value of the right operand and assign the remainder back to left operand. For example, if $a = 20$ , $b = 10 \Rightarrow a \% = b$ will be equal to $a = a \% b$ and therefore, $a = 0$ .
**=	$a ** = b$ will be equal to $a = a ** b$ , for example, if $a = 4$ , $b = 2$ , $a ** = b$ will assign $4 ** 2 = 16$ to $a$ .
//=	$a // = b$ will be equal to $a = a // b$ , for example, if $a = 4$ , $b = 3$ , $a // = b$ will assign $4 // 3 = 1$ to $a$ .

#### 4) Logical Operators

- The logical operators are used primarily in the expression evaluation to make a decision. Python supports the following logical operators.



Operator	Description
and	If both the expression are true, then the condition will be true. If $a$ and $b$ are the two expressions, $a \rightarrow \text{true}$ , $b \rightarrow \text{true} \Rightarrow a \text{ and } b \rightarrow \text{true}$ .
or	If one of the expressions is true, then the condition will be true. If $a$ and $b$ are the two expressions, $a \rightarrow \text{true}$ , $b \rightarrow \text{false} \Rightarrow a \text{ or } b \rightarrow \text{true}$ .
not	If an expression $a$ is true then not ( $a$ ) will be false and vice versa.

#### 5) Bitwise Operators

The bitwise operators perform bit by bit operation on the values of the two operands.

Operator	Description
& (binary and)	If both the bits at the same place in two operands are 1, then 1 is copied to the result. Otherwise, 0 is copied.
(binary or)	The resulting bit will be 0 if both the bits are zero otherwise the resulting bit will be 1.
^ (binary xor)	The resulting bit will be 1 if both the bits are different otherwise the resulting bit will be 0.
~ (negation)	It calculates the negation of each bit of the operand, i.e., if the bit is 0, the resulting bit will be 1 and vice versa.
<< (left shift)	The left operand value is moved left by the number of bits present in the right operand.
>> (right shift)	The left operand is moved right by the number of bits present in the right operand.

For example,

```
if a = 7;  
    b = 6;  
then, binary (a) = 0111  
    binary (b) = 0011
```

```
hence, a & b = 0011  
    a | b = 0111  
    a ^ b = 0100  
    ~ a = 1000
```

## 6) Membership Operators

- Python membership operators are used to check the membership of value inside a data structure.
- If the value is present in the data structure, then the resulting value is true otherwise it returns false.

Operator	Description
in	It is evaluated to be true if the first operand is found in the second operand (list, tuple, or dictionary).
not in	It is evaluated to be true if the first operand is not found in the second operand (list, tuple, or dictionary).

## 7) Identity Operators

Operator	Description
is	It is evaluated to be true if the reference present at both sides point to the same object.
is not	It is evaluated to be true if the reference present at both side do not point to the same object.

## Python Comments

- Comments in Python can be used to explain any program code.
- It can also be used to hide the code as well.
- Comments are the most helpful stuff of any program.
- It enables us to understand the way, a program works.
- In python, any statement written along with **#** symbol is known as a comment.
- The interpreter does not interpret the comment.
- Comment is not a part of the program, but it enhances the interactivity of the program and makes the program readable.
- Python supports two types of comments:

### 1) Single Line Comment:

- In case user wants to specify a single line comment, then comment must start with #

**# This is single line comment.**

```
print "Hello Python"
```

### 2) Multi Line Comment:

Multi lined comment can be given inside triple single quotes.

```
""" This  
Is  
Multiline comment"""
```

```
#single line comment  
print "Hello Python"  
"""This is  
multiline comment"""
```

## Branching programs

- Decision making is the most important aspect of almost all the programming languages.
- Decision making allows us to run a particular block of code for a particular decision.
- The decisions are made on the validity of the particular conditions. Condition checking is the backbone of decision making.
- In python, decision making is performed by the following statements.

Statement	Description
If Statement	The if statement is used to test a specific condition. If the condition is true, a block of code (if-block) will be executed.
If - else Statement	The if-else statement is similar to if statement except the fact that, it also provides the block of the code for the false case of the condition to be checked. If the condition provided in the if statement is false, then the else statement will be executed.
Nested if Statement	Nested if statements enable us to use if ? else statement inside an outer if statement.

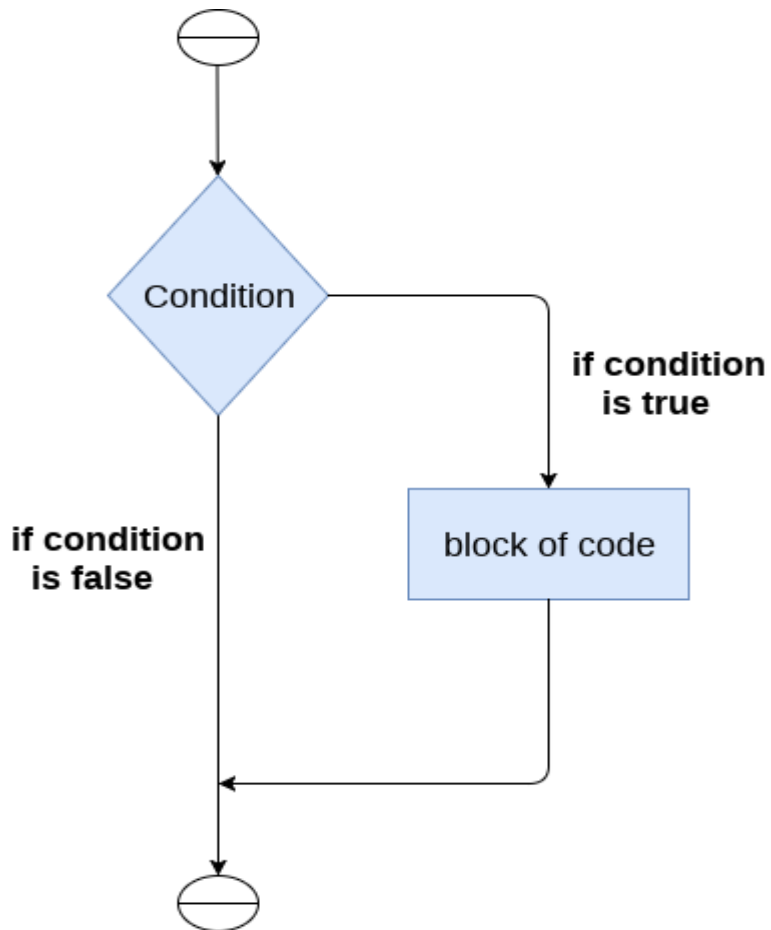
## Indentation in Python

- Python doesn't allow the use of parentheses for the block level code.
- Indentation is used to declare a block.
- If two statements are at the same indentation level then they are the part of the same block.
- Four spaces are given to indent the statements which are a typical amount of indentation in python.
- Indentation is the most used part of the python language since it declares the block of code.
- All the statements of one block are intended at the same level indentation.

## The if statement

- The if statement is used to test a particular condition and if the condition is true, it executes a block of code known as if-block.
- The condition of if statement can be any valid logical expression which can be either evaluated to true or false.





**The syntax of the if-statement is given below.**

```
if expression:  
    statement
```

### **Example**

```
a = int(input("enter the number?"))  
if a % 2 == 0:  
    print("Number is even")
```

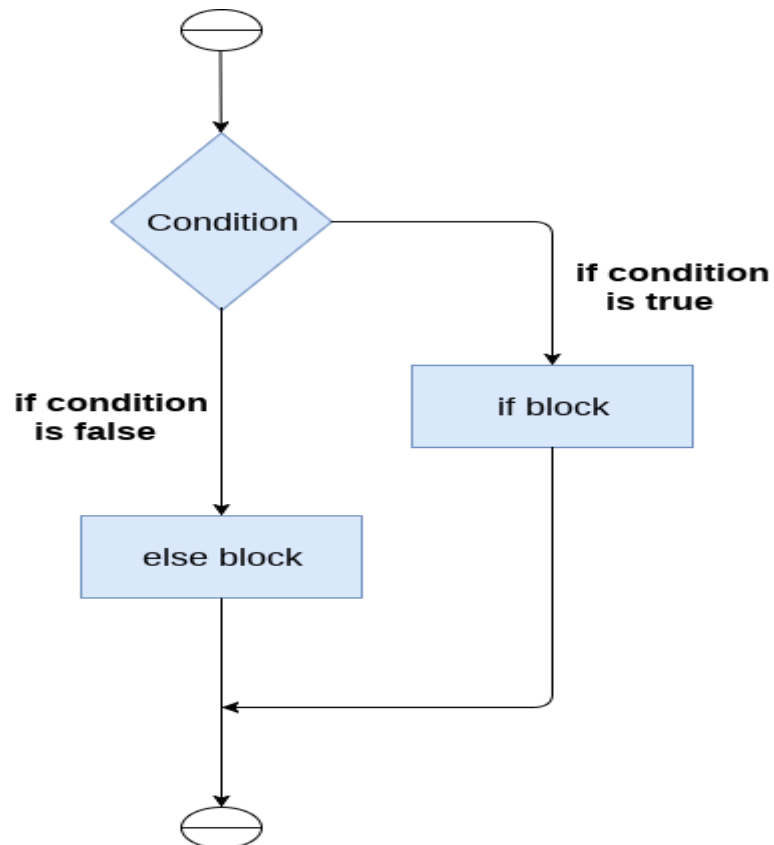
### **Example -2 Program to print the largest of the three numbers.**

```
a = int(input("Enter a? "))  
b = int(input("Enter b? "))  
c = int(input("Enter c? "))  
if a>b and a>c:  
    print("a is largest")  
if b>a and b>c:  
    print("b is largest")  
if c>a and c>b:  
    print("c is largest")
```

### **The if-else statement**

- The if-else statement provides an else block combined with the if statement which is executed in the false case of the condition.

- If the condition is true then the if-block is executed. Otherwise the else-block is executed.



**The syntax of the if-else statement is given below.**

```
if condition:  
    #block of statements  
else:  
    #another block of statements (else-block)
```

**Example to check whether a person is eligible to vote or not.**

```
age = int(input("Enter your age? "))  
if age >= 18:  
    print("You are eligible to vote !!");  
else:  
    print("Sorry! you have to wait !!");
```

## **The elif statement**

- The elif statement enables us to check multiple conditions and execute the specific block of statements depending upon the true condition among them.
- We can have any number of elif statements in our program depending upon our need.
- The elif statement works like an if-else-if ladder statement in C.
- It must be succeeded by an if statement.

The syntax of the elif statement is given below.

**if expression 1:**

**# block of statements**

**elif expression 2:**

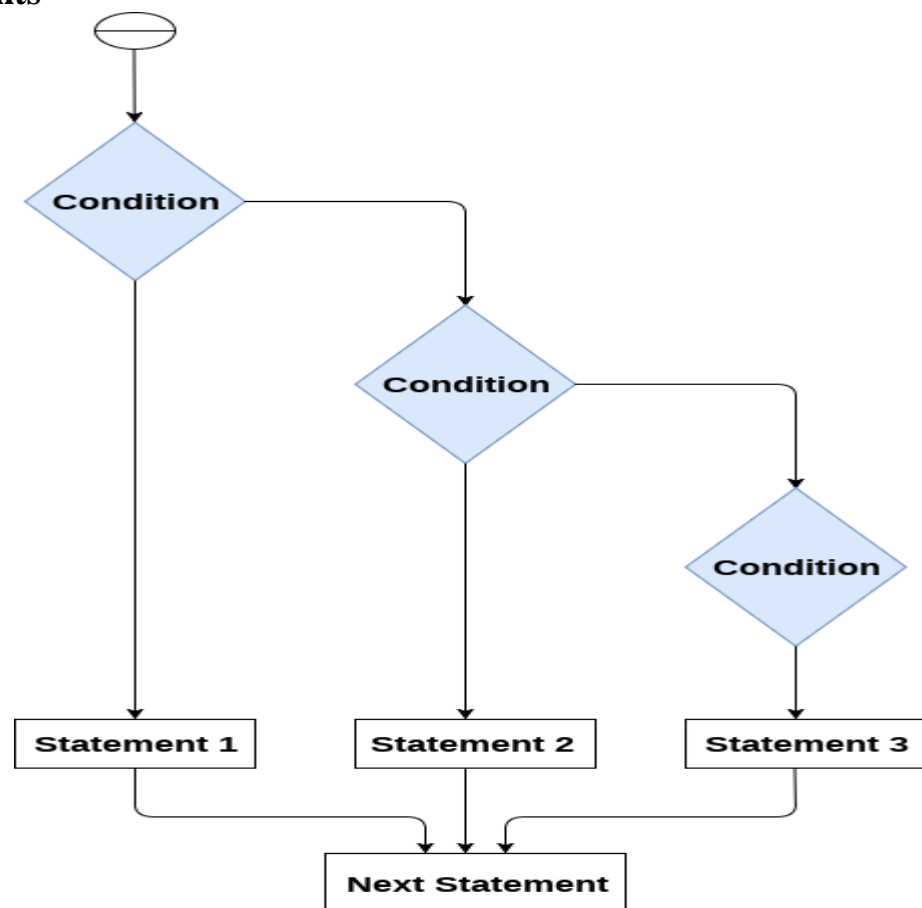
**# block of statements**

**elif expression 3:**

**# block of statements**

**else:**

**# block of statements**



### Example

```
number = int(input("Enter the number?"))
if number==10:
    print("number is equals to 10")
elif number==50:
    print("number is equal to 50");
elif number==100:
    print("number is equal to 100");
else:
    print("number is not equal to 10, 50 or 100");
```

## Strings and Input

- Objects of type str are used to represent strings of characters.
- Literals of type str can be written using either single or double quotes or tripl quotes
- e.g., 'abc' or "abc" or ‘’abc’’
- The literal '123' denotes a string of characters not the number one hundred twenty-three.
- Try typing the following expressions in to the Python interpreter (remember that the >>> is a prompt, not something that you type):

```
'a'  
3*4  
3*'a'  
'a'+'a'
```
- The operator + is said to be overloaded.

## Built-in String functions

- Python provides various in-built functions that are used for string handling.

Method	Description
capitalize()	It capitalizes the first character of the String. This function is deprecated in python3
casefold()	It returns a version of s suitable for case-less comparisons.
center(width ,fillchar)	It returns a space padded string with the original string centred with equal number of left and right spaces.
count(string,begin,end)	It counts the number of occurrences of a substring in a String between begin and end index.
Decode (encoding = 'UTF8', errors = 'strict')	Decodes the string using codec registered for encoding.
encode()	Encode S using the codec registered for encoding. Default encoding is 'utf-8'.
isalnum()	It returns true if the characters in the string are alphanumeric i.e., alphabets or numbers and there is at least 1 character. Otherwise, it returns false.
isalpha()	It returns true if all the characters are alphabets and there is at least one character, otherwise False.
isdecimal()	It returns true if all the characters of the string are decimals.
isdigit()	It returns true if all the characters are digits and there is at least one character, otherwise False.
isidentifier()	It returns true if the string is the valid identifier.
islower()	It returns true if the characters of a string are in lower case,

	otherwise false.
isnumeric()	It returns true if the string contains only numeric characters.
isprintable()	It returns true if all the characters of s are printable or s is empty, false otherwise.
isupper()	It returns false if characters of a string are in Upper case, otherwise False.
isspace()	It returns true if the characters of a string are white-space, otherwise false.

## Input

There are two functions in Python that you can use to read data from the user

- 1) Raw\_Input
- 2) Input

You can store the results from them into a variable.

### 1) Raw\_Input

- raw\_input is used to **read text (strings) from the user**

```
name = raw_input("What is your name? ")
type(name)
```

### 2) Input

- input is used to **read integers**

```
age = input("What is your age? ")
print "Your age is: ", age
type(age)
```

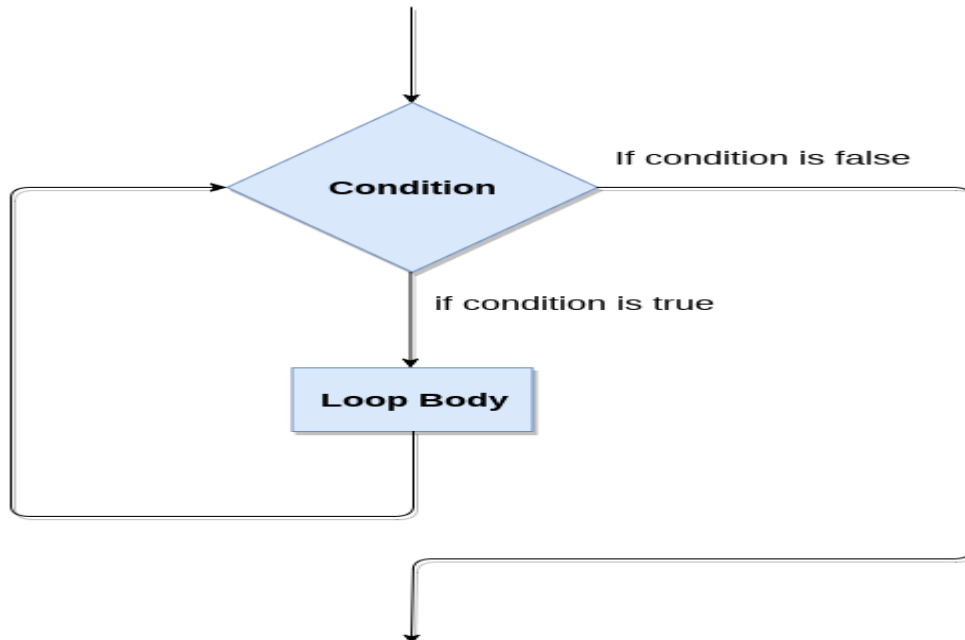
## Iteration

- A generic iteration mechanism is depicted
- If the test evaluates to true the program executes the loop body once and then goes back to reevaluate the test
- This process is repeated until the test evaluates to false after which control passes to the code following the iteration statement

## while loop

- The while loop is also known as a pre-tested loop.
- a while loop allows a part of the code to be executed as long as the given condition is true.
- It can be viewed as a repeating if statement.
- The while loop is mostly used in the case where the number of iterations is not known in advance.

- The syntax is given below  
while expression:  
statements
- The statements can be a single statement or the group of statements.
- The expression should be any valid python expression resulting into true or false.
- The true is any non-zero value.



### Example -1

```

i=1;
while i<=10:
    print(i);
    i=i+1;

```

### Example -2

```

i=1
number=0
b=9
number = int(input("Enter the number?"))
while i<=10:
    print("%d X %d = %d \n"%(number,i,number*i));
    i = i+1;

```

## Python for loop

- The for loop in Python is used to iterate the statements or a part of the program several times.
- It is frequently used to traverse the data structures like list, tuple, or dictionary.
- The syntax of for loop in python is given below.

```

for iterating_var in sequence:
    statement(s)

```

## Example 1

```
i=1
n=int(input("Enter the number up to which you want to print the natural numbers?"))
for i in range(0,10):
    print(i,end = ' ')
```

## Example 2

```
i=1;
num = int(input("Enter a number:"));
for i in range(1,11):
    print("%d X %d = %d"%(num,i,num*i));
```

## Example

### Print each chocolate from a chocolate list:

```
choco = ["Dairy Milk", "Kit kat", "Perk"]
for x in choco:
    print(x)
```

The for loop does not require an indexing variable to set beforehand

## Looping Through a String

Even strings are iterable objects, they contain a sequence of characters:

## Example

Loop through the letters in the word "Apple":

```
for x in "Apple":
    print(x)
```

## The break Statement

With the break statement we can stop the loop before it has looped through all the items:

## Example

Exit the loop when x is "5star":

```
choco = ["Silk", "Bar one", "5star"]
for x in choco:
    if x == "5star":
        break
    print(x)
```

## Python break statement

- The break is a keyword in python which is used to bring the program control out of the loop.
- The break statement breaks the loops one by one.
- In the case of nested loops, it breaks the inner loop first and then proceeds to outer loops.
- we can say that break is used to abort the current execution of the program and the control goes to the next line after the loop.
- The break is commonly used in the cases where we need to break the loop for a given condition.
- The syntax of the break is given below.

```
#loop statements  
break;
```

### Example 1

```
list=[1,2,3,4]  
count = 1;  
for i in list:  
    if i == 4:  
        print("item matched")  
        count = count + 1;  
        break  
    print("found at",count,"location");
```

## Python continue Statement

- The continue statement in python is used to bring the program control to the beginning of the loop.
- The continue statement skips the remaining lines of code inside the loop and start with the next iteration.
- It is mainly used for a particular condition inside the loop so that we can skip some specific code for a particular condition.
- The syntax of Python continue statement is given below.

```
#loop statements  
continue;  
#the code to be skipped
```

### Example 1

```
i = 0;  
while i!=10:  
    print("%d"%i);  
    continue;  
    i=i+1;
```



# Python Functions

- A function is a block of code which only runs when it is called.
- You can pass data, known as parameters, into a function.
- A function can return data as a result.

## Creating a Function

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

## Syntax of function

```
def functionname(formal parameters):  
    Function Body
```

## Example

```
def test (a,b):  
    print("Hello from a function")
```

- Def is a keyword that tells python that a function is defined
- The function name test() is a name that is used to the function
- The sequence of names (a,b ) within the parentheses following the function name are the formal parameters of the function

## Calling a Function / No argument no return value

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

## Example

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

## Parameters with function / (With parameter no return value)

- Information can be passed to functions as parameter.
- Parameters are specified after the function name **inside the parentheses**. Y
- You can **add as many parameters as you want**, just separate them with a comma.
- The following example has a function with one parameter (fname).
- When the function is called we pass along a first name which is used inside the function to print the full name:

## Example

```
def test1(fname):  
    print(fname + " Nanadani")  
  
test1("Addres")  
test1("City")  
test1("Email")
```

## Default Parameter Value

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

## Example

```
def test2(country = "Africa"):  
    print("I am from " + country)  
  
test2("Sweden")  
test2("India")  
test2()  
test2("Brazil")
```

## Return Values

- To let a function return a value, use the return statement:

## Example

```
def test3(x):  
    return 5 * x  
  
print(test3(3))  
print(test3(5))  
print(test3(9))
```

## Scope

- Variables can only reach the area in which they are defined which is called scope.
- Python supports global variables (usable in the entire program) and local variables.
- all variables declared in a function are local variables.
- To access a global variable inside a function it's required to explicitly define 'global variable'
- Below we'll examine the use of local variables and scope. This will not work:

```
def f(x,y):
    print('You called f(x,y) with the value x = ' + str(x) + ' and y = ' + str(y))
    print('x * y = ' + str(x*y))
    z = 4 # cannot reach z, so THIS WON'T WORK
    z = 3
f(3,2)
```

```
def f(x,y):
    z = 3
    print('You called f(x,y) with the value x = ' + str(x) + ' and y = ' + str(y))
    print('x * y = ' + str(x*y))
    print(z) # can reach because variable z is defined in the function
```

```
f(3,2)
```

```
def highFive():
    return 5
```

```
def f(x,y):
    z = highFive() # we get the variable contents from highFive()
    return x+y+z # returns x+y+z. z is reachable because it is defined above
```

```
result = f(3,2)
print(result)
```

## Specification

## Recursion

- 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.

## Python Recursive Function

- a function can call other functions.
- It is even possible for the function to call itself.
- These type of construct are termed as recursive functions.
- Following is an example of 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  $1*2*3*4*5*6 = 720$ .

```
def calc_factorial(x):  
    """This is a recursive function  
    to find the factorial of an integer"""  
  
    if x == 1:  
        return 1  
    else:  
        return (x * calc_factorial(x-1))  
  
num = 4  
print("The factorial of", num, "is", calc_factorial(num))
```

### **Advantages of Recursion**

- 1) Recursive functions make the code look clean and elegant.
- 2) A complex task can be broken down into simpler sub-problems using recursion.
- 3) Sequence generation is easier with recursion than using some nested iteration.

### **Disadvantages of Recursion**

- 1) Sometimes the logic behind recursion is hard to follow through.
- 2) Recursive calls are expensive (inefficient) as they take up a lot of memory and time.
- 3) Recursive functions are hard to debug.

### **Global Variables**

- In Python, a variable declared outside of the function or in global scope is known as global variable.
- Global variable can be accessed inside or outside of the function.
- Let's see an example on how a global variable is created in Python.

```
x = "global"  
def foo():  
    print("x inside :", x)  
foo()  
print("x outside:", x)
```

### **Local Variables**

- A variable declared inside the function's body or in the local scope is known as local variable.

```
def foo():  
    y = "local"  
foo()  
    print(y)
```

- Normally, we declare a variable inside the function to create a local variable.

```
x = "global"  
def foo():  
    global x  
    y = "local"  
    x = x * 2  
    print(x)  
    print(y)  
foo()
```

## What is a Module?

- Consider a module to be the same as a code library.
- A file containing a set of functions you want to include in your application.

## Create a Module

- To create a module just save the code you want in a file with the file extension .py

## Example

Save this code in a file named mymodule.py

```
def greeting(name):  
    print("Hello, " + name)
```

## Use a Module

- Now we can use the module we just created by using the import statement

## Example

Import the module named mymodule and call the greeting function:

```
import mymodule
```

```
mymodule.greeting("Jonathan")
```

**Note:** When using a function from a module, use the syntax: **module\_name.function\_name.**

## Variables in Module

The module can contain functions, as already described, but also variables of all types (arrays, dictionaries, objects etc):

Example

Save this code in the file mymodule.py

```
person1 = {  
    "name": "John",  
    "age": 36,  
    "country": "Norway"  
}
```

### Example

Import the module named mymodule, and access the person1 dictionary:

```
import mymodule
```

```
a = mymodule.person1["age"]  
print(a)
```

## Naming a Module

You can name the module file whatever you like, but it must have the file extension .py

Re-naming a Module

You can create an alias when you import a module, by using the as keyword:

### Example

Create an alias for mymodule called mx:

```
import mymodule as mx
```

```
a = mx.person1["age"]  
print(a)
```

## Built-in Modules

There are several built-in modules in Python, which you can import whenever you like.

Example

Import and use the platform module:

```
import platform
```

```
x = platform.system()  
print(x)
```

## What is a file?

- File is a named location on disk to store related information.
- It is used to permanently store data in a non-volatile memory (e.g. hard disk).
- random access memory (RAM) is volatile which loses its data when computer is turned off, we use files for future use of the data.
- 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 resources that are tied with the file are freed.
- Hence, in Python, a file operation takes place in the following order.
  - Open a file
  - Read or write (perform operation)
  - Close the file

## How to open a file?

- Python has a built-in function `open()` to open a file.
  - The `open()` function takes two parameters; filename, and mode.
  - There are four different methods (modes) for opening a file:
  - "r" - Read - Default value. Opens a file for reading, error if the file does not exist
  - "a" - Append - Opens a file for appending, creates the file if it does not exist
  - "w" - Write - Opens a file for writing, creates the file if it does not exist
  - "x" - Create - Creates the specified file, returns an error if the file exists
- 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:/Python33/README.txt") # specifying full path
```

- We also specify if we want to open the file in text mode or binary mode.
  - You can specify if the file should be handled as binary or text mode
- "t" - Text - Default value. Text mode
- "b" - Binary - Binary mode (e.g. images)

## Syntax

- To open a file for reading it is enough to specify the name of the file:
  - `f = open("demofile.txt")`
- The code above is the same as:
- `f = open("demofile.txt", "rt")`

- Because "r" for read, and "t" for text are the default values, you do not need to specify them.
- Note: Make sure the file exists, or else you will get an error.

### **Example**

```
f = open("demofile.txt", "r")  
print(f.read())
```

### **Read Only Parts of the File**

- By default the read() method returns the whole text, but you can also specify how many character you want to return:

### **Example**

Return the 5 first characters of the file:

```
f = open("demofile.txt", "r")  
print(f.read(5))
```

### **Read Lines**

- You can return one line by using the readline() method:

### **Example**

Read one line of the file:

```
f = open("demofile.txt", "r")  
print(f.readline())
```

By calling readline() two times, you can read the two first lines:

### **Example**

Read two lines of the file:

```
f = open("demofile.txt", "r")  
print(f.readline())  
print(f.readline())
```

By looping through the lines of the file, you can read the whole file, line by line:

### **Example**

Loop through the file line by line:

```
f = open("demofile.txt", "r")  
for x in f:
```



```
print(x)
```

## File Write

### Write to an Existing File

To write to an existing file, you must add a parameter to the `open()` function:

"a" - Append - will append to the end of the file

"w" - Write - will overwrite any existing content

### Example

Open the file "demofile.txt" and append content to the file:

```
f = open("demofile.txt", "a")
f.write("Now the file has one more line!")
```

### Example

Open the file "demofile.txt" and overwrite the content:

```
f = open("demofile.txt", "w")
f.write("Woops! I have deleted the content!")
```

**Note: the "w" method will overwrite the entire file.**

## Create a New File

- To create a new file in Python, use the `open()` method, with one of the following parameters:

"x" - Create - will create a file, returns an error if the file exist

"a" - Append - will create a file if the specified file does not exist

"w" - Write - will create a file if the specified file does not exist

### Example

Create a file called "myfile.txt":

```
f = open("myfile.txt", "x")
```

Result: a new empty file is created!

## Example

Create a new file if it does not exist:

```
f = open("myfile.txt", "w")
```

## Delete a File

To delete a file, you must import the OS module, and run its `os.remove()` function:

## Example

Remove the file "demofile.txt":

```
import os
os.remove("demofile.txt")
Check if File exist:
```

To avoid getting an error, you might want to check if the file exist before you try to delete it:

Example

Check if file exist, then delete it:

```
import os
if os.path.exists("demofile.txt"):
    os.remove("demofile.txt")
else:
    print("The file does not exist")
```

## Delete Folder

To delete an entire folder, use the `os.rmdir()` method:

Example

Remove the folder "myfolder":

```
import os
os.rmdir("myfolder")
```

Note: You can only remove empty folders.

## Tuple

- A tuple is a collection which is ordered and unchangeable.
- In Python tuples are written with round brackets.

Example

## Create a Tuple:

```
thistuple = ("apple", "banana", "cherry")
```

```
print(thistuple)
```

## Access Tuple Items

- You can access tuple items by referring to the index number inside square brackets:

### Example

Return the item in position 1:

```
thistuple = ("apple", "banana", "cherry")  
print(thistuple[1])
```

## Change Tuple Values

- Once a tuple is created, you cannot change its values.
- Tuples are unchangeable.

### Example

You cannot change values in a tuple:

```
thistuple = ("apple", "banana", "cherry")  
thistuple[1] = "blackcurrant"  
# The values will remain the same:  
print(thistuple)
```

## Loop Through a Tuple

- You can loop through the tuple items by using a for loop.

### Example

#### Iterate through the items and print the values:

```
thistuple = ("apple", "banana", "cherry")  
for x in thistuple:  
    print(x)
```

- You will learn more about for loops in our Python For Loops Chapter.

## Check if Item Exists

- To determine if a specified item is present in a tuple use the in keyword:

## Example

Check if "apple" is present in the tuple:

```
thistuple = ("apple", "banana", "cherry")
if "apple" in thistuple:
    print("Yes, 'apple' is in the fruits tuple")
```

## Tuple Length

- To determine how many items a list have, use the len() method:

## Example

Print the number of items in the tuple:

```
thistuple = ("apple", "banana", "cherry")
print(len(thistuple))
```

## Add Items

- Once a tuple is created, you cannot add items to it.
- Tuples are unchangeable.

## Example

You cannot add items to a tuple:

```
thistuple = ("apple", "banana", "cherry")
thistuple[3] = "orange" # This will raise an error
print(thistuple)
```

## Remove Items

- Note: You cannot remove items in a tuple.
- Tuples are unchangeable so you cannot remove items from it but you can delete the tuple completely:

## Example

- The del keyword can delete the tuple completely:

```
thistuple = ("apple", "banana", "cherry")
del thistuple
print(thistuple) #this will raise an error because the tuple no longer exists
```

## The tuple() Constructor

- It is also possible to use the tuple() constructor to make a tuple.

### Example

Using the tuple() method to make a tuple:

```
thistuple = tuple(("apple", "banana", "cherry")) # note the double round-brackets  
print(thistuple)
```

## Tuple Methods

- Python has two built-in methods that you can use on tuples.

Method	Description
count()	Returns the number of times a specified value occurs in a tuple
index()	Searches the tuple for a specified value and returns the position of where it was found
Method	Description

### List

- A list is a collection which is ordered and changeable.
- Python lists are written with square brackets.

### Example

#### Create a List:

```
thislist = ["apple", "banana", "cherry"]  
print(thislist)
```

### Access Items

- You access the list items by referring to the index number:

### Example

Print the second item of the list:

```
thislist = ["apple", "banana", "cherry"]  
print(thislist[1])
```

### Change Item Value

To change the value of a specific item, refer to the index number:

### Example

Change the second item:

```
thislist = ["apple", "banana", "cherry"]  
thislist[1] = "blackcurrant"  
print(thislist)
```

## Loop Through a List

- You can loop through the list items by using a for loop:

### Example

Print all items in the list, one by one:

```
thislist = ["apple", "banana", "cherry"]  
for x in thislist:  
    print(x)
```

- You will learn more about for loops in our Python For Loops Chapter.

## Check if Item Exists

- To determine if a specified item is present in a list use the in keyword:

### Example

- Check if "apple" is present in the list:

```
thislist = ["apple", "banana", "cherry"]  
if "apple" in thislist:  
    print("Yes, 'apple' is in the fruits list")
```

## List Length

- To determine how many items a list have, use the len() method:

### Example

- Print the number of items in the list:  

```
thislist = ["apple", "banana", "cherry"]  
print(len(thislist))
```

## Add Items

- To add an item to the end of the list, use the append() method:

### Example

- Using the append() method to append an item:

```
thislist = ["apple", "banana", "cherry"]
thislist.append("orange")
print(thislist)
```

To add an item at the specified index, use the insert() method:

#### Example

- Insert an item as the second position:

```
thislist = ["apple", "banana", "cherry"]
thislist.insert(1, "orange")
print(thislist)
```

### Remove Item

There are several methods to remove items from a list:

#### Example

The remove() method removes the specified item:

```
thislist = ["apple", "banana", "cherry"]
thislist.remove("banana")
print(thislist)
```

#### Example

The pop() method removes the specified index, (or the last item if index is not specified):

```
thislist = ["apple", "banana", "cherry"]
thislist.pop()
print(thislist)
```

#### Example

The del keyword removes the specified index:

```
thislist = ["apple", "banana", "cherry"]
del thislist[0]
print(thislist)
```

#### Example

The del keyword can also delete the list completely:

```
thislist = ["apple", "banana", "cherry"]  
del thislist  
print(thislist) #this will cause an error because "thislist" no longer exists.
```

### Example

The clear() method empties the list:

```
thislist = ["apple", "banana", "cherry"]  
thislist.clear()  
print(thislist)
```

### The list() Constructor

It is also possible to use the list() constructor to make a list.

### Example

Using the list() constructor to make a List:

```
thislist = list(("apple", "banana", "cherry")) # note the double round-brackets  
print(thislist)
```

### Difference Between List Vs Tuple

List	Tuple
The literal syntax of list is shown by the []	The literal syntax of the tuple is shown by the ()
The List is mutable	The tuple is immutable
The List has the variable length	The tuple has the fixed length
The list provides more functionality than tuple	The tuple provides less functionality than the list
The list Is used in the scenario in which we need to store the simple collections with no constraints where the value of the items can be changed	The tuple is used in the cases where we need to store the read-only collections i.e., the value of the items can not be changed. It can be used as the key inside the dictionary

### Python Classes/Objects

Python is an object oriented programming language.



Almost everything in Python is an object, with its properties and methods.

A Class is like an object constructor, or a "blueprint" for creating objects.

## Create a Class

To create a class, use the keyword class:

Example

Create a class named MyClass, with a property named x:

```
class MyClass:
```

```
    x = 5
```

Create Object

Now we can use the class named myClass to create objects:

Example

Create an object named p1, and print the value of x:

```
p1 = MyClass()
```

```
print(p1.x)
```

## Object Methods

Objects can also contain methods. Methods in objects are functions that belongs to the object.

Let us create a method in the Person class:

Example

Insert a function that prints a greeting, and execute it on the p1 object:

```
class Person:
```

```
    def __init__(self, name, age):
```

```
        self.name = name
```

```
        self.age = age
```

```
    def myfunc(self):
```

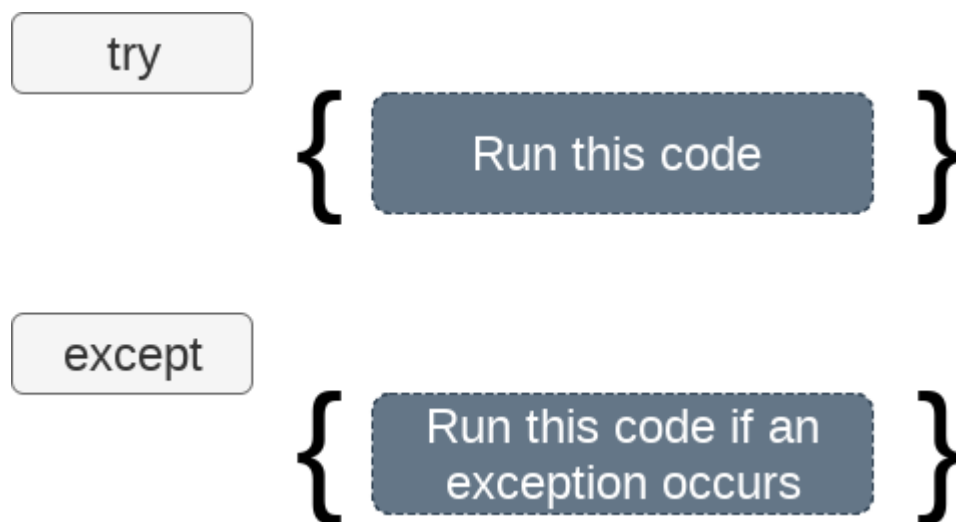
```
        print("Hello my name is " + self.name)
```

```
p1 = Person("John", 36)
```

```
p1.myfunc()
```

## Exception handling in python

- If the python program contains suspicious code that may throw the exception, we must place that code in the try block.
- The try block must be followed with the except statement which contains a block of code that will be executed if there is some exception in the try block.



## Roles of an Exception Handler in Python

### 1) Error handling:

- The exceptions get raised whenever Python detects an error in a program at runtime.
- As a programmer, if you don't want the default behavior then code a 'try' statement to catch and recover the program from an exception.
- Python will jump to the 'try' handler when the program detects an error the execution will be resumed.

### 2) Event Notification:

- Exceptions are also used to signal suitable conditions & then passing result flags around a program & text them explicitly.

### 3) Terminate Execution:

- There may arise some problems or errors in programs that it needs a termination.
- try/finally is used that guarantees that closing-time operation will be performed.
- The 'with' statement offers an alternative for objects that support it.

### 4) Exotic flow of Control:

- Programmers can also use exceptions as a basis for implementing unusual control flow.
- Since there is no 'go to' statement in Python so exceptions can help in this respect.

## Syntax

Here is simple syntax of try....except...else blocks –

```
try:
    You do your operations here
    .....
except ExceptionI:
    If there is ExceptionI, then execute this block.
except ExceptionII:
    If there is ExceptionII, then execute this block.
    .....
else:
    If there is no exception then execute this block.
```

## Simple Program Demostate Exception Handling

### Example -01

```
(a,b) = (6,0)
try:# simple use of try-except block for handling errors
    g = a/b
except ZeroDivisionError:
    print ("This is a DIVIDED BY ZERO error")
```

### Example -02

```
(a,b) = (6,0)
try:
    g = a/b
except ZeroDivisionError as s:
    k = s
    print (k)
#Output will be: integer division or modulo by zero
```

## The try -Except clause with no Exception

- The structure of such type of 'except' clause having no exception is shown with an example

```
try: # all operations are done within this block.
    .....
except: # this block will get executed if any exception encounters.
    .....
else: # this block will get executed if no exception is found.
    .....
```

- All the exceptions get caught where there is try/except the statement of this type.

- Good programmers use this technique of exception to make the program fully executable.

### **Except Clause with multiple Exception**

```
try: # all operations are done within this block.
.....
except ( Exception1 [, Exception2[,....Exception N ] ] ) :
# this block will get executed if any exception encounters from the above lists of
exceptions.
.....
else: # this block will get executed if no exception is found.
```

### **Try-Finally clause**

```
try:
# all operations are done within this block.
.....
# if any exception encounters, this block may get skipped.
finally:
.....
# this block will definitely be executed.
```

### **Example**

This example opens a file, writes content in the, file and comes out gracefully because there is no problem at all –

```
x = "hello"
try:
    x > 3
except NameError:
    print("You have a variable that is not defined.")
except TypeError:
    print("You are comparing values of different type")
```

```
try:
    x = 1/0
except NameError:
    print("You have a variable that is not defined.")
except TypeError:
    print("You are comparing values of different type")
except:
    print("Something else went wrong")
```

```

x = 1
try:
    x > 10
except NameError:
    print("You have a variable that is not defined.")
except TypeError:
    print("You are comparing values of different type")
else:
    print("The 'Try' code was executed without raising any errors!")

```

## Abstract Data Types

- Abstract Data type (ADT) is a type (or class) for objects whose behavior is defined by a set of value and a set of operations.
- The definition of ADT only mentions what operations are to be performed but not how these operations will be implemented.
- It does not specify how data will be organized in memory and what algorithms will be used for implementing the operations.
- It is called “abstract” because it gives an implementation independent view.
- The process of providing only the essentials and hiding the details is known as abstraction.
- The user of data type need not know that data type is implemented, for example, we have been using int, float, char data types only with the knowledge with values that can take and operations that can be performed on them without any idea of how these types are implemented.
- a user only needs to know what a data type can do but not how it will do it.
- We can think of ADT as a black box which hides the inner structure and design of the data type.

## Abstract Classes

- Abstract classes are classes that contain one or more abstract methods.
- An abstract method is a method that is declared but contains no implementation.
- Abstract classes may not be instantiated and require subclasses to provide implementations for the abstract methods.
- Subclasses of an abstract class in Python are not required to implement abstract methods of the parent class.
- Let's look at the following example:

```
import abc
```

```
class AbstractAnimal(object):
    __metaclass__ = abc.ABCMeta
```

```

    @abc.abstractmethod
    def walk(self):
        """ data """

```

```

    @abc.abstractmethod
    def talk(self):
        """ data """

```

```

class Duck(AbstractAnimal):
    name = "

```

```
def __init__(self, name):  
    print('duck created.')  
    self.name = name
```

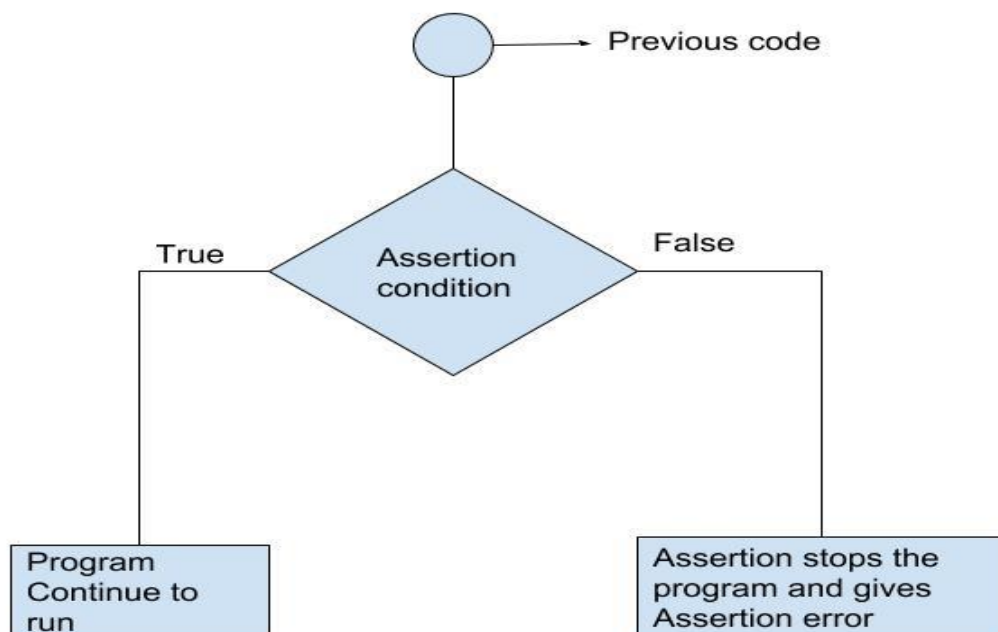
```
def walk(self):  
    print('walks')
```

```
def talk(self):  
    print('quack')
```

```
obj = Duck('duck1')  
obj.talk()  
obj.walk()
```

### What is Assertion?

- Assertions are statements that assert or state a fact confidently in your program.
- For example, while writing a division function, you're confident the divisor shouldn't be zero, you assert divisor is not equal to zero.
- Assertions are simply boolean expressions that checks if the conditions return true or not.
- If it is true, the program does nothing and move to the next line of code.
- if it's false, the program stops and throws an error.
- It is also a debugging tool as it brings the program on halt as soon as any error is occurred and shows on which point of the program error has occurred.
- We can be clear by looking at the flowchart below:



### Python assert Statement

- Python has built-in assert statement to use assertion condition in the program.
- assert statement has a condition or expression which is supposed to be always true.
- If the condition is false assert halts the program and gives an AssertionError.

### Syntax for using Assert in Python:

```
assert <condition>
assert <condition>,<error message>
```

### Example

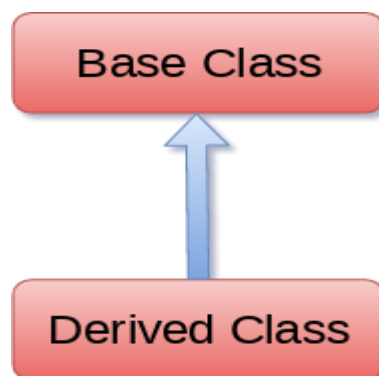
```
x = "hello"
#if condition returns True, then nothing happens:
assert x == "hello"
#if condition returns False, AssertionError is raised:
assert x == "goodbye"
```

### Example

```
x = "hello"
#if condition returns False, AssertionError is raised:
assert x == "goodbye", "x should be 'hello'"
```

## Python Inheritance

- Inheritance is an important aspect of the object-oriented paradigm.
- Inheritance provides code reusability to the program because we can use an existing class to create a new class instead of creating it from scratch.
- the child class acquires the properties and can access all the data members and functions defined in the parent class.
- A child class can also provide its specific implementation to the functions of the parent class. In this section of the tutorial we will discuss inheritance in detail.
- In python, a derived class can inherit base class by just mentioning the base in the bracket after the derived class name.
- Consider the following syntax to inherit a base class into the derived class.



### Syntax

```
class derived-class(base class):
    <class-suite>
```

- A class can inherit multiple classes by mentioning all of them inside the bracket.
- Consider the following syntax.

### Syntax

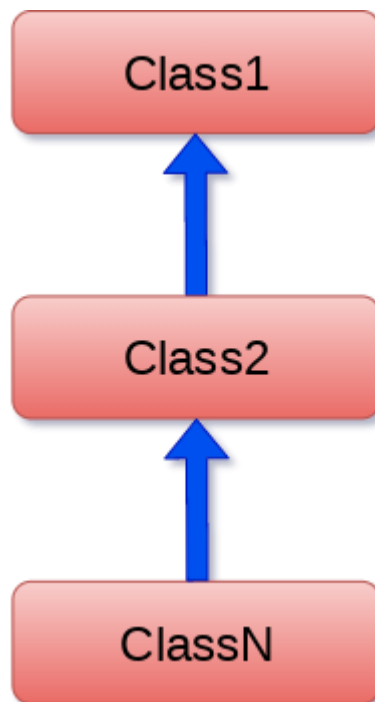
```
class derive-class(<base class 1>, <base class 2>, ..... <base class n>):
    <class - suite>
```

### Example 1

```
class Animal:
    def speak(self):
        print("Animal Speaking")
#child class Dog inherits the base class Animal
class Dog(Animal):
    def bark(self):
        print("dog barking")
d = Dog()
d.bark()
d.speak()
```

### Python Multi-Level inheritance

- Multi-Level inheritance is possible in python like other object-oriented languages.
- Multi-level inheritance is archived when a derived class inherits another derived class.
- There is no limit on the number of levels up to which the multi-level inheritance is archived in python.



**The syntax of multi-level inheritance is given below.**

#### Syntax

```
class class1:
    <class-suite>
class class2(class1):
    <class suite>
class class3(class2):
    <class suite>
```

### Example

```
class Animal:
    def speak(self):
        print("Animal Speaking")
```



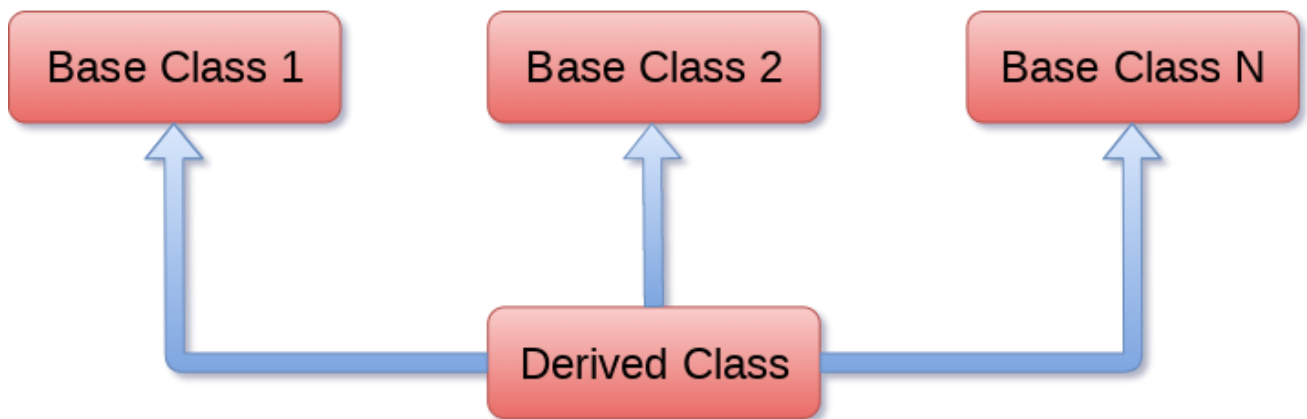
```

#The child class Dog inherits the base class Animal
class Dog(Animal):
    def bark(self):
        print("dog barking")
#The child class Dogchild inherits another child class Dog
class DogChild(Dog):
    def eat(self):
        print("Eating bread...")
d = DogChild()
d.bark()
d.speak()
d.eat()

```

### Python Multiple inheritance

Python provides us the flexibility to inherit multiple base classes in the child class.



The syntax to perform multiple inheritance is given below.

#### Syntax

```

class Base1:
    <class-suite>

class Base2:
    <class-suite>

.
.
.
class BaseN:
    <class-suite>

class Derived(Base1, Base2, ..... BaseN):
    <class-suite>

```

## Example

```
class Calculation1:
    def Summation(self,a,b):
        return a+b;
class Calculation2:
    def Multiplication(self,a,b):
        return a*b;
class Derived(Calculation1,Calculation2):
    def Divide(self,a,b):
        return a/b;
d = Derived()
print(d.Summation(10,20))
print(d.Multiplication(10,20))
print(d.Divide(10,20))
```

## Encapsulation

- Encapsulation is also an important aspect of object-oriented programming.
- It is used to restrict access to methods and variables.
- code and data are wrapped together within a single unit from being modified by accident.
- you can restrict access to methods and variables.
- This can prevent the data from being modified by accident and is known as encapsulation.
- We can say that object-oriented programming relies heavily on encapsulation.
- The terms encapsulation and abstraction (also data hiding) are often used as synonyms.
- They are nearly synonymous
- For Ex. abstraction is achieved through encapsulation.
- Data hiding and encapsulation are the same concept, so it's correct to use them as synonyms.
- Generally speaking encapsulation is the mechanism for restricting the access to some of an object's components, this means that the internal representation of an object can't be seen from outside of the object's definition.
- Access to this data is typically only achieved through special methods: Getters and Setters.
- By using solely get() and set() methods
- we can make sure that the internal data cannot be accidentally set into an inconsistent or invalid state.
- It's nearly always possible to circumvent this protection mechanism:
- E.g. in C++ by the "friends" mechanism in Java and Ruby via reflection API or in Python by name mangling.
- A method to set private data can also be used to do some plausibility checks.
- In our example, we can check if the birthday makes sense e.g. it's not very likely that a customer is more than 100 years old. Or we can rule out that a customer with a giro account is less than 14 years old.

## For Example

```
class Robot(object):
    def __init__(self):
        self.a = 123
        self._b = 123
        self.__c = 123

obj = Robot()
print(obj.a)
```

```
print(obj._b)
print(obj.__c)
```

### Example -2

```
class Car:
    def __init__(self):
        self.__updateSoftware()
    def drive(self):
        print 'driving'
    def __updateSoftware(self):
        print 'updating software'
redcar = Car()
redcar.drive()
```

Type	Description
public methods	Accessible from anywhere
private methods	Accessible only in their own class. starts with two underscores
public variables	Accessible from anywhere
private variables	Accessible only in their own class or by a method if defined. starts with two underscores

### Method overloading example

We create a class with one method sayHello(). The first parameter of this method is set to None, this gives us the option to call it with or without a parameter.

An object is created based on the class, and we call its method using zero and one parameter.  
class Human:

```
def sayHello(self, name=None):
```

```
    if name is not None:
        print 'Hello ' + name
    else:
        print 'Hello '
```

```
# Create instance
obj = Human()
```

```
# Call the method
obj.sayHello()
```

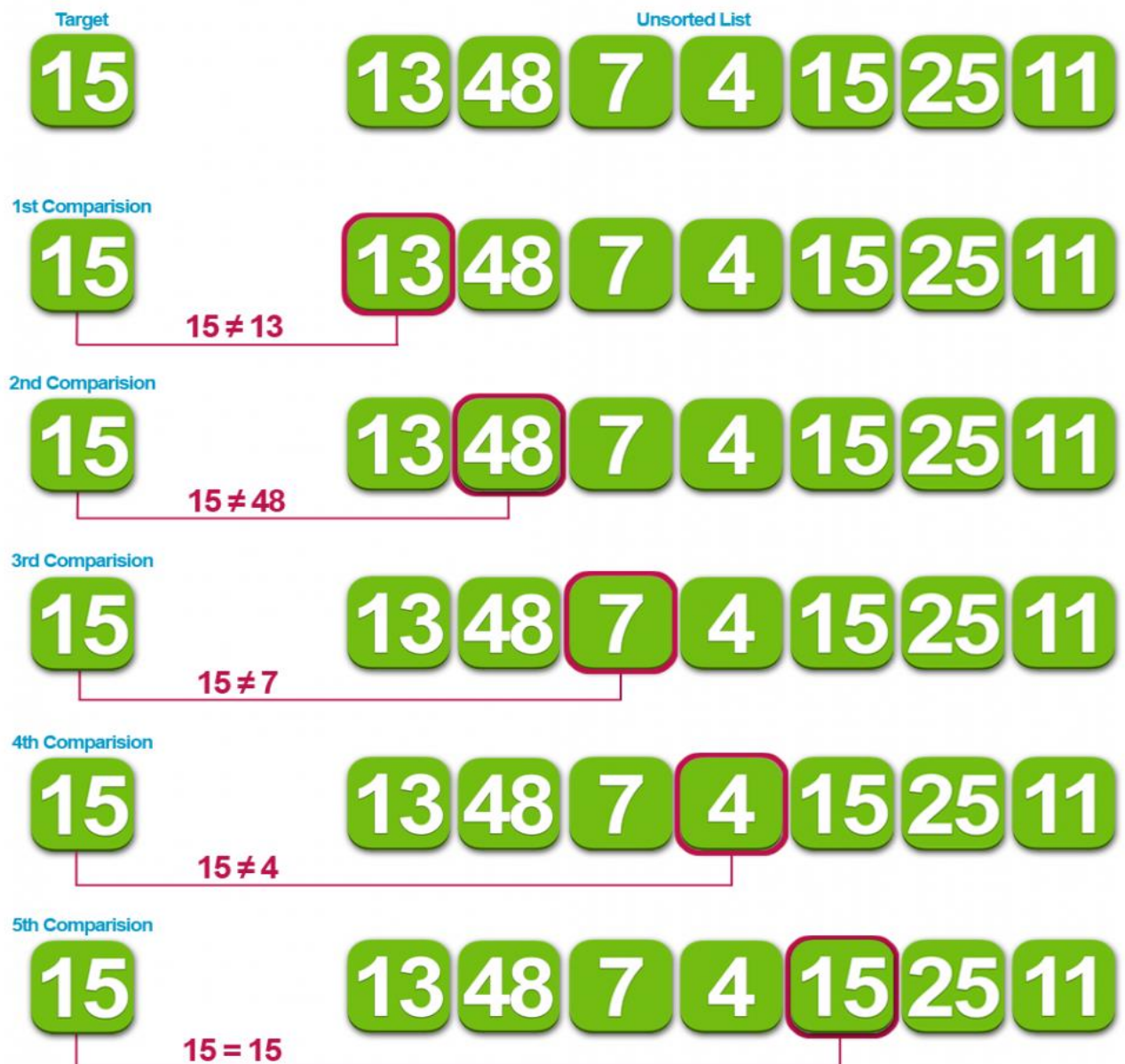
```
# Call the method with a parameter
obj.sayHello("Spkm")
```

## Search Algorithms

- Searching is a very basic necessity when you store data in different data structures.
- The simplest approach is to go across every element in the data structure and match it with the value you are searching for.
- This is known as Linear search.
- It is inefficient and rarely used but creating a program for it gives an idea about how we can implement some advanced search algorithms.

### 1) Linear Search

- Linear search is one of the simplest searching algorithm in which targeted item in sequentially matched with each item in a list.
- It is poorest searching algorithm with poorest case time complexity  $O(n)$ .



```
list_of_elements = [4, 2, 8, 9, 3, 7]
```

```
x = int(input("Enter number to search: "))  
found = False
```

```
for i in range(len(list_of_elements)):  
    if(list_of_elements[i] == x):  
        found = True  
        print("%d found at %dth position"%(x,i))  
        break
```

```
if(found == False):  
    print("%d is not in list"%x)
```

## 2 Binary Search

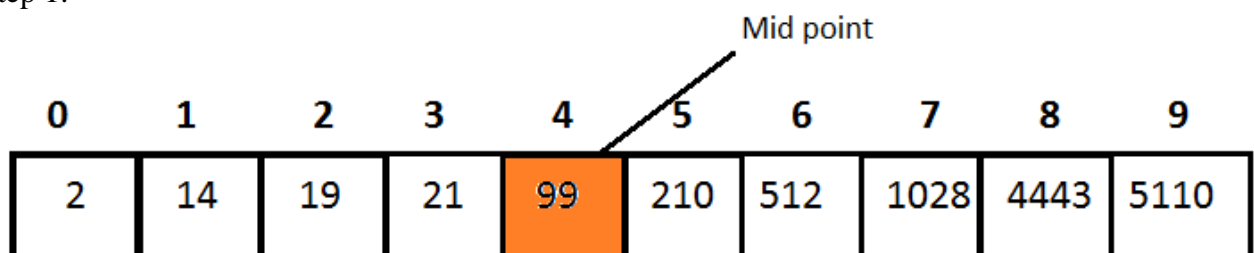
- In linear search, we have to check each node/element.
- Because of this, time complexity increases.
- To reduce this time complexity we use Binary search.
- In Binary search half of the given array will be ignored after just one comparison.
- The main point to be noted is Binary Search only works for sorted array.
- If the array is sorted into ascending order, all we have to do is find the middle index of the array and then compare the element at middle index with the element to find.
- If our given element is greater than the element at middle index then we'll ignore the left half array and the array will start from the next index of middle index.
- if the given element is less than the element presents at middle index, then we'll ignore the right half array and the new array will ends with left one index of middle index.
- If the given element is equal to the element presents on the middle index, then search is completed.
- the first index is greater than the last index of array, it means we have been gone through the entire array and the given element is not presented in the array.

### Python Binary Search

Example:

We've a sorted array [2, 14, 19, 21, 99, 210, 512, 1028, 4443, 5110] and the element to be find is 4443.

Step 1:



0	1	2	3	4	5	6	7	8	9
2	14	19	21	99	210	512	1028	4443	5110

starting index = 0, last index = 9, mid = (starting index + last index) / 2 = 9 / 2 = 4.5 = 4

Step 2:

4443 > 99

0	1	2	3	4	5	6	7	8	9
					210	512	1028	4443	5110

starting index = mid+1 = 5, last index = 9, mid = (9+5)/2 = 7

Step 3:

4443 > 1028

0	1	2	3	4	5	6	7	8	9
								4443	5110

starting index = mid + 1 = 7+1 = 8, last index = 9, mid = (9+8)/2 = 8

Step-4

4443 = 4443

element at middle index = element to be find

0	1	2	3	4	5	6	7	8	9
								4443	5110

Element is present at index 8

```
# Python code to demonstrate working  
# of binary search in library  
from bisect import bisect_left
```

```
def BinarySearch(a, x):  
    i = bisect_left(a, x)  
    if i != len(a) and a[i] == x:  
        return i  
    else:  
        return -1
```

```
a = [1, 2, 4, 4, 8]  
x = int(4)  
res = BinarySearch(a, x)  
if res == -1:  
    print(x, "is absent")  
else:  
    print("First occurrence of", x, "is present at", res)
```

## Sorting Algorithms

- Sorting refers to arranging data in a particular format.
- Sorting algorithm specifies the way to arrange data in a particular order.
- Most common orders are in numerical or lexicographical order.
- Sorting is also used to represent data in more readable formats.
- Below we see five such implementations of sorting in python.

- 1) Bubble Sort
- 2) Merge Sort
- 3) Insertion Sort
- 4) Shell Sort
- 5) Selection Sort

### 1. Bubble Sort

- It is a comparison-based algorithm in which each pair of head-to-head elements is compared and the elements are swapped if they are not in order.

```
def bubblesort(list):  
# Swap the elements to arrange in order  
    for iter_num in range(len(list)-1,0,-1):  
        for idx in range(iter_num):  
            if list[idx]>list[idx+1]:  
                temp = list[idx]  
                list[idx] = list[idx+1]  
                list[idx+1] = temp  
  
list = [19,2,31,45,6,11,121,27]  
bubblesort(list)  
print(list)
```

### 2 Merge Sort

- Merge sort first divides the array into equal halves and then combines them in a sorted manner.

```
def merge_sort(unsorted_list):  
    if len(unsorted_list) <= 1:  
        return unsorted_list  
# Find the middle point and divide it  
    middle = len(unsorted_list) // 2  
    left_list = unsorted_list[:middle]  
    right_list = unsorted_list[middle:]  
  
    left_list = merge_sort(left_list)  
    right_list = merge_sort(right_list)  
    return list(merge(left_list, right_list))  
  
# Merge the sorted halves  
  
def merge(left_half, right_half):
```

```

res = []
while len(left_half) != 0 and len(right_half) != 0:
    if left_half[0] < right_half[0]:
        res.append(left_half[0])
        left_half.remove(left_half[0])
    else:
        res.append(right_half[0])
        right_half.remove(right_half[0])
if len(left_half) == 0:
    res = res + right_half
else:
    res = res + left_half
return res

```

```

unsorted_list = [64, 34, 25, 12, 22, 11, 90]

```

```

print(merge_sort(unsorted_list))

```

### 3. Insertion Sort

- Insertion sort involves finding the right place for a given element in a sorted list.
- so in beginning we compare the first two elements and sort them by comparing them.
- Then we pick the third element and find its proper position among the previous two sorted elements.
- This way we gradually go on adding more elements to the already sorted list by putting them in their proper position.

```

def insertion_sort(InputList):
    for i in range(1, len(InputList)):
        j = i-1
        nxt_element = InputList[i]
        # Compare the current element with next one

        while (InputList[j] > nxt_element) and (j >= 0):
            InputList[j+1] = InputList[j]
            j=j-1
        InputList[j+1] = nxt_element

```

```

list = [19,2,31,45,30,11,121,27]
insertion_sort(list)
print(list)

```

### 4. Selection Sort

- In selection sort we start by finding the minimum value in a given list and move it to a sorted list.
- Then we repeat the process for each of the remaining elements in the unsorted list.
- The next element entering the sorted list is compared with the existing elements and placed at its correct position.
- So at the end all the elements from the unsorted list are sorted.

```

def selection_sort(input_list):

```



```

for idx in range(len(input_list)):

    min_idx = idx
    for j in range( idx +1, len(input_list)):
        if input_list[min_idx] > input_list[j]:
            min_idx = j
    # Swap the minimum value with the compared value

    input_list[idx], input_list[min_idx] = input_list[min_idx], input_list[idx]

```

```

l = [19,2,31,45,30,11,121,27]
selection_sort(l)
print(l)

```

#### 4) Shell Sort

Shell sort is a highly efficient sorting algorithm and is based on insertion sort algorithm. This algorithm avoids large shifts as in case of insertion sort, if the smaller value is to the far right and has to be moved to the far left.

This algorithm uses insertion sort on a widely spread elements, first to sort them and then sorts the less widely spaced elements. This spacing is termed as interval. This interval is calculated based on Knuth's formula as –  
Knuth's Formula

$$h = h * 3 + 1$$

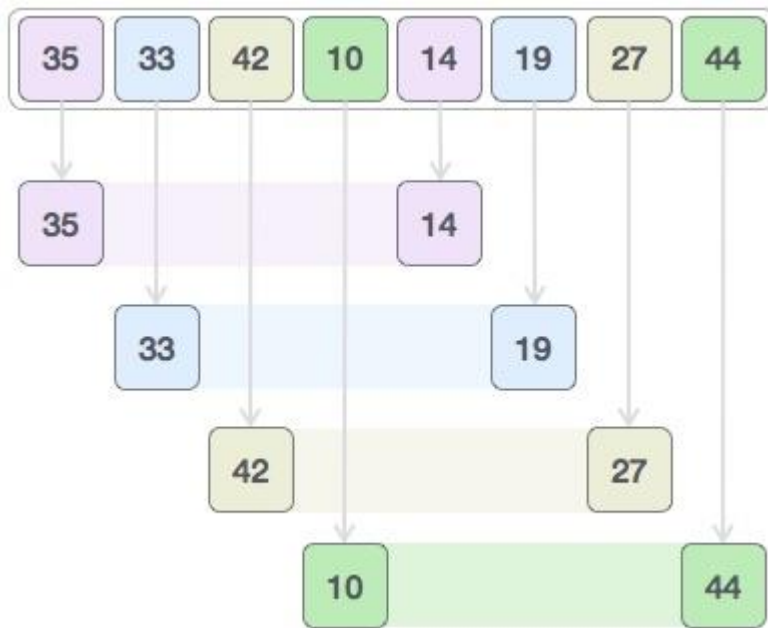
where –

h is interval with initial value 1

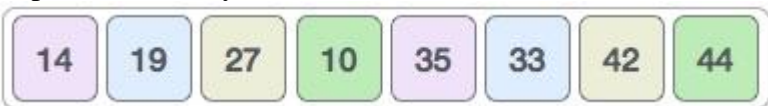
This algorithm is quite efficient for medium-sized data sets as its average and worst-case complexity of this algorithm depends on the gap sequence the best known is  $O(n)$ , where n is the number of items. And the worst case space complexity is  $O(n)$ .

How Shell Sort Works?

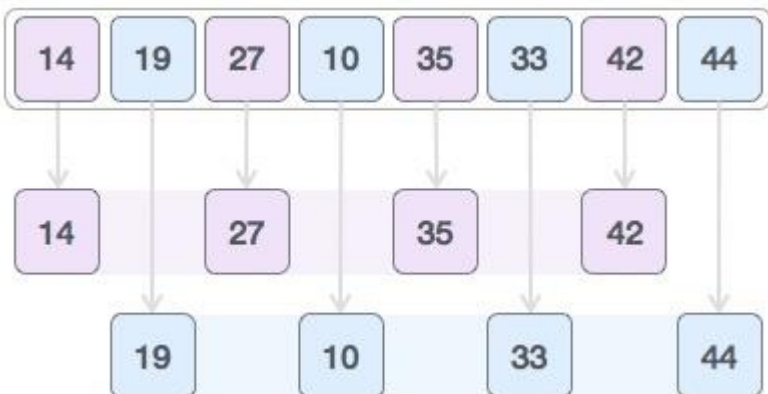
Let us consider the following example to have an idea of how shell sort works. We take the same array we have used in our previous examples. For our example and ease of understanding, we take the interval of 4. Make a virtual sub-list of all values located at the interval of 4 positions. Here these values are {35, 14}, {33, 19}, {42, 27} and {10, 44}



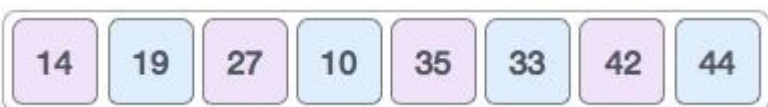
We compare values in each sub-list and swap them (if necessary) in the original array. After this step, the new array should look like this –



Then, we take interval of 1 and this gap generates two sub-lists - {14, 27, 35, 42}, {19, 10, 33, 44}



We compare and swap the values, if required, in the original array. After this step, the array should look like this –



Finally, we sort the rest of the array using interval of value 1. Shell sort uses insertion sort to sort the array.

Following is the step-by-step depiction –



## Hash tables

- Hash tables are a type of data structure in which the address or the index value of the data element is generated from a hash function.
- That makes accessing the data faster as the index value behaves as a key for the data value.
- In other words Hash table stores key-value pairs but the key is generated through a hashing function.
- the search and insertion function of a data element becomes much faster as the key values themselves become the index of the array which stores the data.
- In Python, the Dictionary data types represent the implementation of hash tables.
- The Keys in the dictionary satisfy the following requirements.

- The keys of the dictionary are hashable i.e. they are generated by a hashing function which generates a unique result for each unique value supplied to the hash function.
- The order of data elements in a dictionary is not fixed.
- we see the implementation of a hash table by using the dictionary data types as below.
- When we talk about hash tables
- we're actually talking about a dictionary.
- While an array can be used to construct hash tables, arrays index their elements using integers.
- if we want to store data and use keys other than integers, such as 'string',
- we may want to use a dictionary.
  
- Dictionaries in Python are implemented using hash tables.
- It is an array whose indexes are obtained using a hash function on the keys.
- We declare an empty dictionary like this:  
`D = {}`

Then, we can add its elements:

```
D['a'] = 1
D['b'] = 2
D['c'] = 3
D{'a': 1, 'c': 3, 'b': 2}
```

- It's a structure with (key, value) pair:

`D[key] = value`

- The string used to "index" the hash table D is called the "key".
- To access the data stored in the table, we need to know the key:

`D['b']` 2

How we loop through the hash table?

```
for k in D.keys():
    ... print D[k]
...
1
3
2
```

- If we want to print the (key, value) pair:

```
for k,v in D.items():
    ... print k,':',v
...
a : 1
c : 3
b : 2
```

## Unit - 3 Plotting using PyLab

### Plotting using PyLab

- PyLab is a Python standard library module that provides many of the facilities of MATLAB
- matplotlib.pyplot is a collection of command style functions that make matplotlib work like MATLAB.
- “a high-level technical computing language and interactive environment for algorithm development, data visualization, data analysis, and numeric computation.”
- Each pyplot function makes some change to a figure: e.g., creates a figure, creates a plotting area in a figure, plots some lines in a plotting area, decorates the plot with labels, etc.
- In matplotlib.pyplot various states are preserved across function calls, so that it keeps track of things like the current figure and plotting area, and the plotting functions are directed to the current axes (please note that “axes” here and in most places in the documentation refers to the axes part of a figure and not the strict mathematical term for more than one axis).

#### Example 1

```
import matplotlib.pyplot as plt
plt.plot([1,2,3,4])
plt.ylabel('some numbers')
plt.show()
```

If you provide a single list or array to the plot() command, matplotlib assumes it is a sequence of y values, and automatically generates the x values for you.

python ranges start with 0, the default x vector has the same length as y but starts with 0. Hence the x data are [0,1,2,3].

**plot() is a versatile command, and will take an arbitrary number of arguments For example, to plot x versus y, you can issue the command like plt.plot([1, 2, 3, 4], [1, 4, 9, 16])**

- For every x, y pair of arguments, there is an **optional third argument** which is the format string that indicates the **color and line type of the plot**.

- The letters and symbols of the format string are from MATLAB, and you concatenate a color string with a line style string.
- The **default format string is 'b-'**, which is a solid blue line.
- For example, to plot the above with red circles, you would issue

### Example 2

```
import matplotlib.pyplot as plt
plt.plot([1,2,3,4], [1,4,9,16], 'ro')
plt.axis([0, 6, 0, 20])
plt.show()
```

**axis() command in the example above takes a list of [xmin, xmax, ymin, ymax] and specifies the viewport of the axes.**

**The axis() command in the example above takes a list of [xmin, xmax, ymin, ymax] and specifies the viewport of the axes.**

- If matplotlib were limited to working with lists, it would be fairly useless for numeric processing.
- you will use numpy arrays.
- all sequences are converted to numpy arrays internally.
- The example below illustrates a plotting several lines with different format styles in one command using arrays.

### Example 3

```
import matplotlib.pyplot as plt
fig = plt.figure()
ax = fig.add_subplot(111)
ax.plot([1, 2, 3, 4], [10, 20, 25, 30], color='pink', linewidth=3)
ax.scatter([0.3, 3.8, 1.2, 2.5], [11, 25, 9, 26], color='red', marker='^')
ax.set_xlim(0.5, 4.5)
plt.show()
```

### Example 4

```
import pylab
principal = 10000 #initial investment
interestRate = 0.05
years = 20
values = []
for i in range(years + 1):
    values.append(principal)
principal += principal*interestRate
pylab.plot(values)
pylab.show()
```

[http://jakevdp.github.io/mpl\\_tutorial/tutorial\\_pages/tut1.html](http://jakevdp.github.io/mpl_tutorial/tutorial_pages/tut1.html)

- The bar at the top contains the name of the window.
- The middle section of the window contains the plot generated by the invocation of **pylab.plot**.
- The two parameters of `pylab.plot` must be sequences of the same length.
- The first specifies the **x-coordinates** of the points to be plotted, and
- the second specifies the **y-coordinates**.
- Together, they provide a sequence of four <x, y> coordinate pairs, [(1,1), (2,7), (3,3), (4,5)].
- These are plotted in order. As each point is plotted, a line is drawn connecting it to the previous point.
- The final line of code, **pylab.show()**, causes the window to appear on the computer screen.
- If that line were not present, the figure would still have been produced, but it would **not have been displayed**.
- This is not as silly as it at first sounds, since one might well choose to write a figure directly to a file, as we will do later, rather than display it on the screen.
- The bar at the bottom of the window contains a number of push buttons.
- The rightmost button is used to write the plot to a file.
- The next button to the left is used to adjust the appearance of the plot in the window.
- The next four buttons are used for panning and zooming. And the button on the left is used to restore the figure to its original appearance after you are done playing with pan and zoom.

- It is possible to produce multiple figures and to write them to files.
- These files can have any name you like, but they will all have the file extension .png.
- The file extension .png indicates that the file is in the **Portable Networks Graphics** format.

## Plotting mortgages and extended examples

### What is a Mortgage?

- A mortgage is a loan in which property or real estate is used as collateral.
- The borrower enters into an agreement with the lender (usually a bank) wherein the borrower receives cash upfront then makes payments over a set time span until he pays back the lender in full.
- A mortgage is often referred to as home loan when its used for the purchase of a home.

### What are the different types of mortgages?

There are two main types of mortgages:

- **Fixed rate:** The interest you're charged stays the same for a number of years, typically between two to five years.
- **Variable rate:** The interest you pay can change
- **Fixed rate mortgages**
  - The interest rate you pay will stay the same throughout the length of the deal no matter what happens to interest rates.
  - You'll see them advertised as 'two-year fix' or 'five-year fix', for example, along with the interest rate charged for that period.

### Variable rate mortgages



- the interest rate can change at any time.
- Make sure you have some savings set aside so that you can afford an increase in your payments if rates do rise.
- Variable rate mortgages come in various forms:

### **Standard variable rate (SVR)**

- This is the normal interest rate your mortgage lender charges homebuyers and it will last as long as your mortgage or until you take out another mortgage deal.
- Changes in the interest rate might occur after a rise or fall in the base rate set by the Bank of England.

### **Discount mortgages**

- This is a discount off the lender's standard variable rate (SVR) and only applies for a certain length of time, typically two or three years.
- But it pays to shop around. SVRs differ across lenders, so don't assume that the bigger the discount, the lower the interest rate.

### **Tracker mortgages**

- Tracker mortgages move directly in line with another interest rate – normally the Bank of England's base rate plus a few percent.
- So if the base rate goes up by 0.5%, your rate will go up by the same amount.
- Usually they have a short life, typically two to five years, though some lenders offer trackers which last for the life of your mortgage or until you switch to another deal.

## Simple Example of Mortgage

“our program should be producing plots designed to show how the mortgage behaves over time.”

- class Mortgage by adding methods that make it convenient to produce such plots.
- The methods plotPayments and plotBalance are simple one-liners, but they do use
- a form of pylab.plot that we have not yet seen. When a figure contains multiple plots, it is useful to produce a key that identifies what each plot is intended to represent.
- each invocation of pylab.plot uses the label keyword
- argument to associate a string with the plot produced by that invocation. (This and other keyword arguments must follow any format strings.)
- A key can then be added to the figure by calling the function pylab.legend
- The nontrivial methods in class Mortgage are plotTotPd and plotNet.
- The method plotTotPd simply plots the cumulative total of the payments made.
- The method plotNet plots an approximation to the total cost of the mortgage over time by plotting the cash expended minus the equity acquired by paying off part of the loan

```
import matplotlib.pyplot as plot
class Mortgage(object):
    #Abstract class for building different kinds of mortgages
    def __init__(self, loan, annRate, months):
        #Create a new mortgage"""
```

```
self.loan = loan
self.rate = annRate/12.0
self.months = months
self.paid = [0.0]
self.owed = [loan]
self.payment = findPayment(loan, self.rate, months)
self.legend = None #description of mortgage
```

```
def makePayment(self):
    #Make a payment
    self.paid.append(self.payment)
    reduction = self.payment - self.owed[-1]*self.rate
    self.owed.append(self.owed[-1] - reduction)
```

```
def getTotalPaid(self):
    #Return the total amount paid so far
    return sum(self.paid)
```

```
def __str__(self):
    return self.legend
```

```
def plotPayments(self, style):
    pylab.plot(self.paid[1:], style, label = self.legend)
```

```
def plotBalance(self, style):
    pylab.plot(self.owed, style, label = self.legend)
```

```
def plotTotPd(self, style):
```

```
    #Plot the cumulative total of the payments made
    totPd = [self.paid[0]]
```

```

for i in range(1, len(self.paid)):
    totPd.append(totPd[-1] + self.paid[i])
    pylab.plot(totPd, style, label = self.legend)

```

```

def plotNet(self, style):
    #Plot an approximation to the total cost of the mortgage over time by plotting the cash
    expended minus the equity acquired by paying off part of the loan
    totPd = [self.paid[0]]
    for i in range(1, len(self.paid)):
        totPd.append(totPd[-1] + self.paid[i])
        #Equity acquired through payments is amount of original loan # paid to date, which is
        amount of loan minus what is still owed
        equityAcquired = pylab.array([self.loan]*len(self.owed))
        equityAcquired = equityAcquired - pylab.array(self.owed)
        net = pylab.array(totPd) - equityAcquired
        pylab.plot(net, style, label = self.legend)

```

```

plot.show()

```

## Fibonacci Sequences, Revisited

➤ a straightforward recursive implementation of the Fibonacci function

### Example 5

```

def fib(n):
    #"""Assumes n is an int >= 0 Returns Fibonacci of n"""
    if n == 0 or n == 1:
        return 1
    else:
        return fib(n-1) + fib(n-2)

```

- 
- ```
graph TD; fib6["fib(6)"] --> fib5["fib(5)"]; fib6 --> fib4_1["fib(4)"]; fib5 --> fib4_2["fib(4)"]; fib5 --> fib3_1["fib(3)"]; fib4_1 --> fib3_2["fib(3)"]; fib4_1 --> fib2_1["fib(2)"]; fib4_2 --> fib3_3["fib(3)"]; fib4_2 --> fib2_2["fib(2)"]; fib3_1 --> fib2_3["fib(2)"]; fib3_1 --> fib1_1["fib(1)"]; fib3_2 --> fib2_4["fib(2)"]; fib3_2 --> fib1_2["fib(1)"]; fib2_1 --> fib1_3["fib(1)"]; fib2_1 --> fib0_1["fib(0)"]; fib2_2 --> fib1_4["fib(1)"]; fib2_2 --> fib0_2["fib(0)"]; fib2_3 --> fib1_5["fib(1)"]; fib2_3 --> fib0_3["fib(0)"]; fib2_4 --> fib1_6["fib(1)"]; fib2_4 --> fib0_4["fib(0)"]; fib1_1 --> fib0_5["fib(0)"]; fib1_2 --> fib0_6["fib(0)"]; fib1_3 --> fib0_7["fib(0)"]; fib1_4 --> fib0_8["fib(0)"]; fib1_5 --> fib0_9["fib(0)"]; fib1_6 --> fib0_10["fib(0)"];
```

➤ Notice that we are computing the same values over and over again.

- For example fib gets called with 3 three times, and each of these calls provokes four additional calls of fib.
- It doesn't require a genius to think that it might be a good idea to record the value returned by the first call, and then look it up rather than compute it each time it is needed.
- This is called **memoization**,
- The function **fastFib** has a parameter, **memo**, that it uses to keep track of the numbers it has already evaluated.
- The parameter has a default value, the empty dictionary, so that clients of fastFib don't have to worry about supplying an initial value for memo.
- When **fastFib** is called with an **n > 1**, it attempts to look up n in memo.
- If it is not there (because this is the first time fastFib has been called with that value), an exception is raised.
- When this happens, fastFib uses the normal Fibonacci recurrence, and then stores the result in memo.

```
def fastFib(n, memo = {}):
    # Assumes n is an int >= 0, memo used only by recursive calls Returns Fibonacci of n
    if n == 0 or n == 1:
        return 1
    try:
        return memo[n]
    except KeyError:
        result = fastFib(n-1, memo) + fastFib(n-2, memo)
        memo[n] = result
    return result
```

- a memo If you try running fastFib, you will see that it is indeed quite fast: fib(120) returns almost instantly.
- What is the complexity of fastFib? It calls fib exactly once for each value from 0 to n. Therefore, under the assumption that dictionary lookup can be done in constant time, the time complexity of fastFib(n) is  $O(n)$ .120

## Dynamic Programming and the 0/1 Knapsack Problem

- One of the optimization problems was the 0/1 knapsack problem.
- Recall that we looked at a greedy algorithm that ran in  $n \log n$  time, but was not guaranteed to find an optimal solution.
- We also looked at a brute-force algorithm that was guaranteed to find an optimal solution, but ran in exponential time.
- Finally, we discussed the fact that the problem is inherently exponential in the size of the input.
- In the worst case, one cannot find an optimal solution without looking at all possible answers.
- the situation is not as bad as it looks.
- **Dynamic programming provides a practical method for solving most 0/1 knapsack problems in a reasonable amount of time.**
- First step in deriving such a solution
- we begin with an exponential solution based on exhaustive enumeration.
- The key idea is to think about exploring the space of possible solutions by constructing a rooted binary tree that enumerates all states that satisfy the weight constraint.

### **A rooted binary tree is an acyclic directed graph in which**

- There is exactly one node with no parents. This is called the root.
- Each non-root node has exactly one parent.
- Each node has at most two children. A childless node is called a leaf.
- Each node in the search tree for the 0/1 knapsack problem is labeled with a multiply that denotes a partial solution to the knapsack problem.

### **The elements of the quadruple are:**

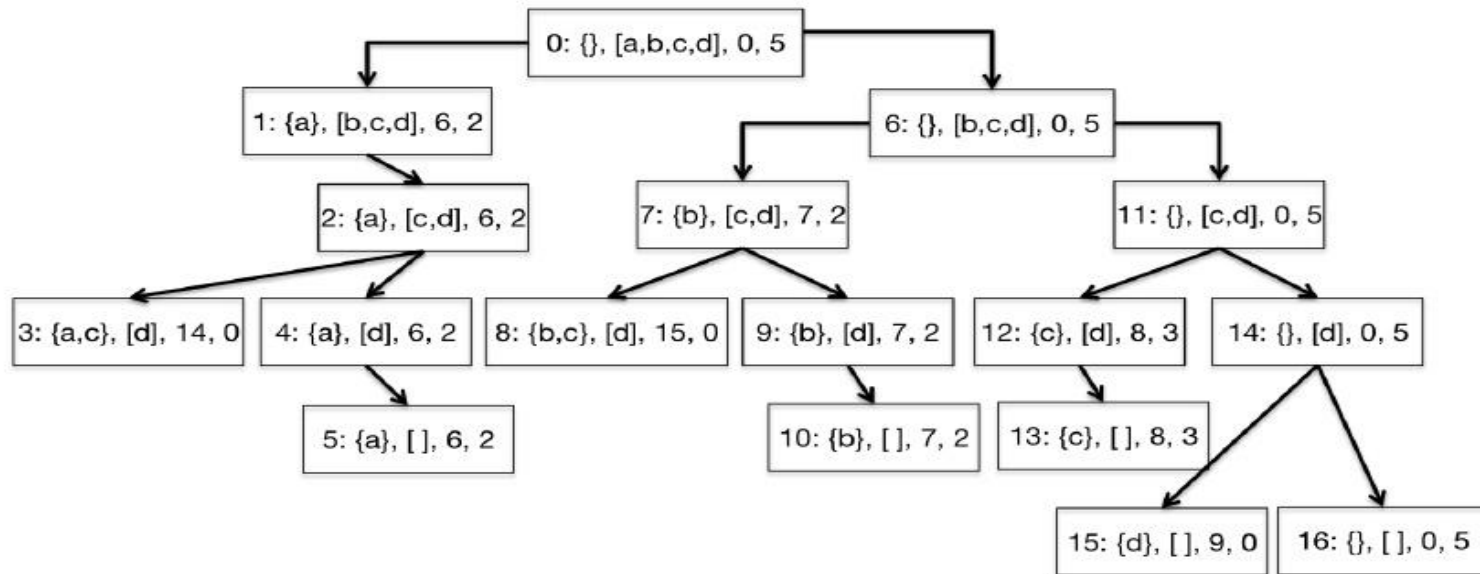
- A set of items to be taken
- The list of items for which a decision has not been made,
- The total value of the items in the set of items to be taken (this is merely an optimization, since the value could be computed from the set), and

- The remaining space in the knapsack. (Again, this is an optimization since it is merely the difference between the weight allowed and the weight of all the items taken so far.)
- The tree is built top-down starting with the root.
- One element is selected from the still-to-be-considered items.
- If there is room for that item in the knapsack,
- a node is constructed that reflects the consequence of choosing to take that item.
- we draw that node as the left child. The right child shows the consequences of choosing not to take that item.
- The process is then applied recursively until either the knapsack is full or there are no more items to consider.
- Because each edge represents a decision (to take or not to take an item), such trees are called decision trees.
- In given table is a table describing a set of items.
- a decision tree for deciding which of those items to take under the assumption that the knapsack has a maximum weight of 5.

| <b>Name</b> | <b>Value</b> | <b>Weight</b> |
|-------------|--------------|---------------|
| A           | 6            | 3             |
| B           | 7            | 3             |
| C           | 8            | 2             |
| D           | 9            | 5             |

**Table of items with values and weights**





**Decision tree for knapsack problem**

- The root of the tree (node 0) has a label  $\langle \{ \}, [a,b,c,d], 0, 5 \rangle$ , indicating that no items have been taken all items remain to be considered the value of the items taken is 0, and a weight of 5 is still available.
- Node 1 indicates that a has been taken, [b,c,d] remain to be considered the value of the items taken is 6 and the knapsack can hold another 2 pounds.
- There is no node to the left of node 1, since item b which weighs 3 pounds, would not fit in the knapsack.
- In Above Example, the numbers that precede the colon in each node indicate one order in which the nodes could be generated.
- This particular ordering is called left-first depth-first.
- At each node we attempt to generate a left node.
- If that is impossible, we attempt to generate a right node.
- If that too is impossible, we back up one node (to the parent) and repeat the process.
- we find ourselves having generated all descendants of the root, and the process halts.

- When the process halts, each combination of items that could fit in the knapsack has been generated, and any leaf node with the greatest value represents an optimal solution.
- Notice that for each leaf node, either the second element is the empty list (indicating that there are no more items to consider taking) or the fourth element is 0 (indicating that there is no room left in the knapsack).
- Unsurprisingly (especially if you read the previous chapter), the natural implementation of a depth-first tree search is recursive.
- Figure 18.6 contains such an implementation.
- It uses class Item from Figure 17.2. The function maxVal returns two values, the set of items chosen and the total value of those items.
- It is called with two arguments, corresponding to the second and fourth elements of the labels of the nodes in the tree:
- toConsider. Those items that nodes higher up in the tree (corresponding to earlier calls in the recursive call stack) have not yet considered.
- avail. The amount of space still available.
- Notice that the implementation of maxVal does not build the decision tree and then look for an optimal node.
- it uses the local variable result to record the best solution found so far.

```
def maxVal(toConsider, avail):
```

```
# Assumes toConsider a list of items, avail a weight Returns a tuple of the total weight of a
#solution to the 0/1 knapsack problem and the items of that solution"""
```

```
if toConsider == [] or avail == 0:
```

```
    result = (0, ())
```

```
elif toConsider[0].getWeight() > avail:
```

```
    #Explore right branch only
```

```
    result = maxVal(toConsider[1:], avail)
```

```
else:
```

```
nextItem = toConsider[0]
#Explore left branch
withVal, withToTake = maxVal(toConsider[1:],
    avail - nextItem.getWeight())
withVal += nextItem.getValue()
#Explore right branch withoutVal, withoutToTake = maxVal(toConsider[1:],avail)
#Choose better branch
if withVal > withoutVal:
    result = (withVal, withToTake + (nextItem,))
else:
    result = (withoutVal, withoutToTake)
return result
def smallTest():
    names = ['a', 'b', 'c', 'd']
    vals = [6, 7, 8, 9]
    weights = [3, 3, 2, 5]
    Items = []
    for i in range(len(vals)):
        Items.append(Item(names[i], vals[i], weights[i]))
    val, taken = maxVal(Items, 5)
    for item in taken:
        print (item)
    print( 'Total value of items taken =', val)
```

## **Dynamic programming and divide and conquer**

- Like divide-and-conquer algorithms, dynamic programming is based upon solving independent subproblems and then combining those solutions.
- The important differencee Divide-and-conquer algorithms are based upon finding subproblems that are substantially smaller than the original problem.
- For example, merge sort works by dividing the problem size in half at each step.
- Dynamic programming involves solving problems that are only slightly smaller than the original problem.
- For example, computing the 19th Fibonacci number is not a substantially smaller problem than computing the 20th Fibonacci number. Another important distinction is that the efficiency of divide-and-conquer algorithms does not depend upon structuring the algorithm
- the same problems are solved repeatedly.
- Dynamic programming is efficient only when the number of distinct subproblems is significantly smaller than the total number of subproblems.

## UNIT-4 Regular Expressions

- Regular expressions are used to identify whether a pattern exists in a given sequence of characters (string) or not.
- They help in manipulating textual data, which is often a pre-requisite for data science projects that involve text mining.
- You must have come across some application of regular expressions
- They are used at the server side to validate the format of email addresses or password during registration, used for parsing text data files to find, replace or delete certain string, etc.
- Regular expressions are extremely powerful
- Searches are accomplished using the `search()` function or method and matching is done with the `match()` function or method.
- REs are strings containing text and special characters which describe a pattern with which to recognize multiple strings.
- We also briefly discussed a regular expression alphabet and for general text, the alphabet used for regular expressions is the set of all **uppercase and lowercase letters plus numeric digits**.
- Specialized alphabets are also possible, for instance, one consisting of only the characters "0" and "1."
- The set of all strings over this alphabet describes all binary strings, i.e., "0," "1," "00," "01," "10," "11," "100," etc.
- REs are sometimes considered an "advanced topic," they can also be rather simplistic

| RE Pattern | String(s) Matched |
|------------|-------------------|
| foo        | foo               |
| Python     | Python            |
| abc123     | abc123            |

- The power of regular expressions comes in when special characters are used to define character sets, subgroup matching, and pattern repetition
- It is these special symbols that allow a RE to match a set of strings rather than a single one

### The most common uses of regular expressions are:

- 1) Search a string (search and match)
- 2) Finding a string (findall)
- 3) Break string into a sub strings (split)
- 4) Replace part of a string (sub)

Let's look at the methods that library "re" provides to perform these tasks.

### What are various methods of Regular Expressions?

- The 're' package provides multiple methods to perform queries on an input string. Here are the most commonly used methods, I will discuss:
  - 1) `re.match()`
  - 2) `re.search()`
  - 3) `re.findall()`
  - 4) `re.split()`
  - 5) `re.sub()`
  - 6) `re.compile()`

Let's look at them one by one.

## 1) re.match(pattern, string):

### Example -1

- This method finds match if it occurs at start of the string.
- For example, calling match() on the string 'AV Analytics AV' and looking for a pattern 'AV' will match.
- if we look for only Analytics, the pattern will not match
- Use **“r”** at the start of the pattern string, it designates a python **raw string**

```
import re
result = re.match(r'AV', 'AV Analytics Vidhya AV')
print (result)
```

➤

### Example -2

- it shows that pattern match has been found.
- To print the matching string we'll use method group (It helps to return the matching string).

```
import re
result = re.match(r'AV', 'AV Analytics Vidhya AV')
print (result.group(0))
```

### Example -3

- Now find 'Analytics' in the given string.
- we see that string is not starting with 'AV' so it should return no match.

```
import re
result = re.match(r'Analytics', 'AV Analytics Vidhya AV')
print (result)
```

### Example -4

- There are methods like start() and end() to know the start and end position of matching pattern in the string.
- you can see that start and end position of matching pattern 'AV' in the string and sometime it helps a lot while performing manipulation with the string

```
import re
result = re.match(r'AV', 'AV Analytics Vidhya AV')
print (result.start())
print (result.end())
```

## 2) re.search(pattern, string):

### Example -1

- It is similar to match() but it doesn't restrict us to find matches at the beginning of the string only.
- Unlike previous method, here searching for pattern 'Analytics' will return a match.

```
import re
result = re.search(r'Analytics', 'AV Analytics Vidhya AV')
print (result.group(0))
```

### **re.search(pattern, string):**

| Example -1                                                                                                                                                                                                                                        |
|---------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------|
| <ul style="list-style-type: none"><li>➤ It is similar to match() but it doesn't restrict us to find matches at the beginning of the string only.</li><li>➤ previous method, here searching for pattern 'Analytics' will return a match.</li></ul> |
| <pre>result = re.search(r'Analytics', 'AV Analytics Vidhya AV') print (result.group(0))</pre>                                                                                                                                                     |
| <ul style="list-style-type: none"><li>➤ you can see that, search() method is able to find a pattern from any position of the string but it only returns the first occurrence of the search pattern</li></ul>                                      |

### **re.findall (pattern, string):**

| Example -1                                                                                                                                                                                                                                                                                                                                                                                                    |
|---------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------|
| <ul style="list-style-type: none"><li>➤ It helps to get a list of all matching patterns. It has no constraints of searching from start or end.</li><li>➤ If we will use method findall to search 'AV' in given string it will return both occurrence of AV.</li><li>➤ While searching a string, I would recommend you to use re.findall() always, it can work like re.search() and re.match() both.</li></ul> |
| <pre>result = re.findall(r'AV', 'AV Analytics Vidhya AV') print (result)</pre>                                                                                                                                                                                                                                                                                                                                |
| <ul style="list-style-type: none"><li>➤ you can see that, search() method is able to find a pattern from any position of the string but it only returns the first occurrence of the search pattern</li></ul>                                                                                                                                                                                                  |

### **re.split(pattern, string, [maxsplit=0]):**

| Example -1                                                                                                                                                                                                                                                                                                |
|-----------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------|
| <ul style="list-style-type: none"><li>➤ This methods helps to split string by the occurrences of given pattern.</li></ul>                                                                                                                                                                                 |
| <pre>result=re.split(r'y','Analytics') result</pre>                                                                                                                                                                                                                                                       |
| <ul style="list-style-type: none"><li>➤ we have split the string "Analytics" by "y". Method split() has another argument "maxsplit".</li><li>➤ It has default value of zero.</li><li>➤ it does the maximum splits that can be done, but if we give value to maxsplit, it will split the string.</li></ul> |

|                                                                                                                                                                                                |
|------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------|
| Example                                                                                                                                                                                        |
| <pre>result=re.split(r'i','Analytics Vidhya') print (result)</pre>                                                                                                                             |
| Example                                                                                                                                                                                        |
| <pre>result=re.split(r'i','Analytics Vidhya',maxsplit=1) print(result)</pre>                                                                                                                   |
| <ul style="list-style-type: none"><li>➤ you can notice that we have fixed the maxsplit to 1.</li><li>➤ the result is, it has only two values whereas first example has three values.</li></ul> |

### **re.sub(pattern, repl, string):**

| Example -1                                                                                                                                                                            |
|---------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------|
| <ul style="list-style-type: none"><li>➤ It helps to search a pattern and replace with a new sub string.</li><li>➤ If the pattern is not found, string is returned unchanged</li></ul> |
| <pre>result=re.sub(r'India','the World','AV is largest Analytics community of India') print(result)</pre>                                                                             |

## re.compile(pattern, repl, string):

### Example -1

- We can combine a regular expression pattern into pattern objects
- which can be used for pattern matching.
- It also helps to search a pattern again without rewriting it.

```
import re
pattern=re.compile('AV')
result=pattern.findall('AV Analytics Vidhya AV')
print result
result2=pattern.findall('AV is largest analytics community of India')
print result2
```

## Regexes and Python

- Python currently supports regular expressions through the re module, which was introduced way back in ancient history
- (Python 1.5), replacing the deprecated regex and reesub modules—both modules were removed from Python in version 2.5, and importing either module from that release on triggers an ImportError exception
- The re module supports the more powerful and regular Perl-style (Perl 5) regexes, allows multiple threads to share the same compiled regex objects, and supports named subgroups
- The more popular functions and methods from the re module.
- Many of these functions are also available as methods of compiled regular expression objects (regex objects and regex match objects
- the two main functions/methods, match() and search(), as well as the compile() function

| Function/Method           | Description                                                             |
|---------------------------|-------------------------------------------------------------------------|
| re Module Function Only   |                                                                         |
| compile(pattern, flags=0) | Compile regex pattern with any optional flags and return a regex object |

## re Module Functions and Regex Object Methods

|                                      |                                                                                                                                                                                                      |
|--------------------------------------|------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------|
| match(pattern, string, flags=0)      | Attempt to match pattern to string with optional flags; return match object on success, None on failure                                                                                              |
| search(pattern, string, flags=0)     | Search for first occurrence of pattern within string with optional flags; return match object on success, None on failure                                                                            |
| findall(pattern, string[, flags])a   | Look for all (non-overlapping) occurrences of pattern in string; return a list of matches                                                                                                            |
| finditer(pattern, string[, flags])   | Same as findall(), except returns an iterator instead of a list; for each match, the iterator returns a match object                                                                                 |
| split(pattern, string, max=0)        | Split string into a list according to regex pattern delimiter and return list of successful matches, splitting at most max times (split all occurrences is the default)                              |
| sub(pattern, repl, string, count=0)c | Replace all occurrences of the regex pattern in string with repl, substituting all occurrences unless count provided (see also subn(), which, in addition, returns the number of substitutions made) |
| purge()                              | Purge cache of implicitly compiled regex patterns                                                                                                                                                    |



## A Longer Regex example

Now will run through an in-depth example of the different ways to use regular expressions for string manipulation.

The first step is to come up with some code that actually generates random (but not too random) data on which to operate.

a script that generates a data set. Although this program simply displays the generated set of strings to standard output, this output could very well be redirected to a test file.

## Data Generator for Regex Exercises

This script creates random data for regular expressions practice and outputs the generated data to the screen.

just convert print to a function, switch from xrange() back to range(), and change from using sys.maxint to sys.maxsize.

```
from random import randrange, choice
from string import ascii_lowercase as lc
from sys import maxint
from time import ctime
tlds = ('com', 'edu', 'net', 'org', 'gov')
for i in xrange(randrange(5, 11)):
    dtint = randrange(maxint) # pick date
    dtstr = ctime(dtint) # date string
    llen = randrange(4, 8) # login is shorter
    login = ''.join(choice(lc) for j in range(llen))
    dlen = randrange(llen, 13) # domain is longer
    dom = ''.join(choice(lc) for j in xrange(dlen))
    print ('%s::%s@%s.%s::%d-%d-%d') % (dtstr, login, dom, choice(tlds), dtint, llen, dlen)
```

## Matching a String

## Common Regular Expression Attributes

## Python RegEx

- A RegEx, or Regular Expression, is a sequence of characters that forms a search pattern.
- RegEx can be used to check if a string contains the specified search pattern.

## RegEx Module

- Python has a built-in package called re, which can be used to work with Regular Expressions.

Import the re module:

```
import re
```

## RegEx in Python

When you have imported the re module, you can start using regular expressions:

## Example

Search the string to see if it starts with "The" and ends with "Spain":

```
import re
```

```
txt = "The rain in Spain"
```

```
x = re.search("^The.*Spain$", txt)
```

## Metacharacters

Metacharacters are characters with a special meaning:

| Character | Description                                                                | Example       |
|-----------|----------------------------------------------------------------------------|---------------|
| []        | A set of characters                                                        | "[a-m]"       |
| \         | Signals a special sequence (can also be used to escape special characters) | "\d"          |
| .         | Any character (except newline character)                                   | "he..o"       |
| ^         | Starts with                                                                | "^hello"      |
| \$        | Ends with                                                                  | "world\$"     |
| *         | Zero or more occurrences                                                   | "aix*"        |
| +         | One or more occurrences                                                    | "aix+"        |
| { }       | Exactly the specified number of occurrences                                | "al{2}"       |
|           | Either or                                                                  | "falls stays" |
| ()        | Capture and group                                                          |               |

```
import re
str = "The rain in Spain"
#Find all lower case characters alphabetically between "a" and "m":
x = re.findall("[a-m]", str)
print(x)
```

```
import re
str = "That will be 59 dollars"
#Find all digit characters:
x = re.findall("\d", str)
print(x)
```

```
import re
str = "hello world"
#Search for a sequence that starts with "he", followed by two (any) characters, and an "o":
x = re.findall("he..o", str)
print(x)
```

```
import re
str = "hello world"
#Check if the string starts with 'hello':
x = re.findall("^hello", str)
if (x):
    print("Yes, the string starts with 'hello'")
else:
    print("No match")
```

```
import re
str = "hello world"
#Check if the string ends with 'world':
x = re.findall("world$", str)
```

|                                                                                                                                                                                                                                                                                |
|--------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------|
| <pre> if (x):     print("Yes, the string ends with 'world'") else:     print("No match") </pre>                                                                                                                                                                                |
| <pre> import re str = "The rain in Spain falls mainly in the plain!" #Check if the string contains "ai" followed by 0 or more "x" characters: x = re.findall("aix*", str) print(x) if (x):     print("Yes, there is at least one match!") else:     print("No match") </pre>   |
| <pre> import re str = "The rain in Spain falls mainly in the plain!" #Check if the string contains "ai" followed by 1 or more "x" characters: x = re.findall("aix+", str) print(x) if (x):     print("Yes, there is at least one match!") else:     print("No match") </pre>   |
| <pre> import re str = "The rain in Spain falls mainly in the plain!" #Check if the string contains "a" followed by exactly two "l" characters: x = re.findall("al{2}", str) print(x) if (x):     print("Yes, there is at least one match!") else:     print("No match") </pre> |
| <pre> import re str = "The rain in Spain falls mainly in the plain!" #Check if the string contains either "falls" or "stays": x = re.findall("falls stays", str) print(x) if (x):     print("Yes, there is at least one match!") else:     print("No match") </pre>            |

### Special Sequences (Special Character)

A special sequence is a \ followed by one of the characters in the list below, and has a special meaning:

| Character | Description                                                                                                    | Example                   |
|-----------|----------------------------------------------------------------------------------------------------------------|---------------------------|
| \A        | Returns a match if the specified characters are at the beginning of the string                                 | "\AThe"                   |
| \b        | Returns a match where the specified characters are at the beginning or at the end of a word                    | r"ain\b"    r"\bain"      |
| \B        | Returns a match where the specified characters are present, but NOT at the beginning (or at the end) of a word | r"\Bain"    or   r"ain\B" |
| \d        | Returns a match where the string contains digits (numbers from 0-9)                                            | "\d"                      |
| \D        | Returns a match where the string DOES NOT contain digits                                                       | "\D"                      |
| \s        | Returns a match where the string contains a white space                                                        | "\s"                      |

|    |                                                                                                                                         |           |
|----|-----------------------------------------------------------------------------------------------------------------------------------------|-----------|
|    | character                                                                                                                               |           |
| \S | Returns a match where the string DOES NOT contain a white space character                                                               | "\S"      |
| \w | Returns a match where the string contains any word characters (characters from a to Z, digits from 0-9, and the underscore _ character) | "\w"      |
| \W | Returns a match where the string DOES NOT contain any word characters                                                                   | "\W"      |
| \Z | Returns a match if the specified characters are at the end of the string                                                                | "Spain\Z" |

```
import re
str = "The rain in Spain"
#Check if the string starts with "The":
x = re.findall("\AThe", str)
print(x)
if (x):
    print("Yes, there is a match!")
else:
    print("No match")
```

```
import re
str = "The rain in Spain"
#Check if "ain" is present at the beginning of a WORD:
x = re.findall(r"\bain", str)
print(x)
if (x):
    print("Yes, there is at least one match!")
else:
    print("No match")
```

```
import re
str = "The rain in Spain"
#Check if "ain" is present at the beginning of a WORD:
x = re.findall(r"\bain", str)
print(x)
if (x):
    print("Yes, there is at least one match!")
else:
    print("No match")
```

```
import re
str = "The rain in Spain"
#Check if "ain" is present, but NOT at the beginning of a word:
x = re.findall(r"\Bain", str)
print(x)
if (x):
    print("Yes, there is at least one match!")
else:
    print("No match")
```

```
import re
str = "The rain in Spain"
#Check if "ain" is present, but NOT at the end of a word:
x = re.findall(r"ain\b", str)
print(x)
if (x):
    print("Yes, there is at least one match!")
else:
```

|                                                                                                                                                                                                                                                                                         |
|-----------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------|
| print("No match")                                                                                                                                                                                                                                                                       |
| <pre>import re str = "The rain in Spain" #Check if the string contains any digits (numbers from 0-9): x = re.findall("\d", str) print(x) if (x):     print("Yes, there is at least one match!") else:     print("No match")</pre>                                                       |
| <pre>import re str = "The rain in Spain" #Return a match at every no-digit character: x = re.findall("\D", str) print(x) if (x):     print("Yes, there is at least one match!") else:     print("No match")</pre>                                                                       |
| <pre>import re str = "The rain in Spain" #Return a match at every white-space character: x = re.findall("\s", str) print(x) if (x):     print("Yes, there is at least one match!") else:     print("No match")</pre>                                                                    |
| <pre>import re str = "The rain in Spain" #Return a match at every word character (characters from a to Z, digits from 0-9, and the underscore _ character): x = re.findall("\w", str) print(x) if (x):     print("Yes, there is at least one match!") else:     print("No match")</pre> |
| <pre>import re str = "The rain in Spain" #Return a match at every NON word character (characters NOT between a and Z. Like "!", "?" white-space etc.): x = re.findall("\W", str) print(x) if (x):     print("Yes, there is at least one match!") else:     print("No match")</pre>      |
| <pre>import re str = "The rain in Spain" #Check if the string ends with "Spain": x = re.findall("Spain\Z", str) print(x) if (x):     print("Yes, there is a match!")</pre>                                                                                                              |

```
else:
    print("No match")
```

## Charcter Sets

A set is a set of characters inside a pair of square brackets [] with a special meaning:

| Set        | Description                                                                                                            |
|------------|------------------------------------------------------------------------------------------------------------------------|
| [arn]      | Returns a match where one of the specified characters (a, r, or n) are present                                         |
| [a-n]      | Returns a match for any lower case character, alphabetically between a and n                                           |
| [^arn]     | Returns a match for any character EXCEPT a, r, and n                                                                   |
| [0123]     | Returns a match where any of the specified digits (0, 1, 2, or 3) are present                                          |
| [0-9]      | Returns a match for any digit between 0 and 9                                                                          |
| [0-5][0-9] | Returns a match for any two-digit numbers from 00 and 59                                                               |
| [a-zA-Z]   | Returns a match for any character alphabetically between a and z, lower case OR upper case                             |
| [+]        | In sets, +, *, .,  , (), \$, {} has no special meaning, so [+] means: return a match for any + character in the string |

## What is Text Processing ?

- what type of applications you create, inevitably, you will need to process human-readable data, which is referred to generally as text
- Python's standard library provides three text processing modules and packages to help you

- 1) CSV(Comma Sepearated values)
- 2) JSON (JavaScript Object Notation)
- 3) XML (Extensible Markup Language )

### 1) CSV(Comma Sepearated values)

- It is a common way to move data into and out of spreadsheet(Excel Sheet) applications in plain text
- It like as opposed to a proprietary binary file format
- It doesn't represent true structured data
- The contents of CSV files are just rows of string values delimited by commas
- You actually don't need the power of a CSV-oriented module.
- Data, writing CSV out to a file, and then reading the same data back

```
import csv
from distutils.log import warn as printf
DATA = (
    (1, 'Introducation', 'Abc'),
    (2, 'Details of Chapter', 'xyz,pqr,mano'),
    (3, 'Exercise', 'Hello,thi is sample'))
printf('*** WRITING CSV DATA')
f = open('bookdata.csv', 'w')
writer = csv.writer(f)
for record in DATA:
    writer.writerow(record)
f.close()

printf('*** REVIEW OF SAVED DATA')
f = open('bookdata.csv', 'r')
reader = csv.reader(f)
for chap in reader:
    printf('Chapter %s:',(chap))
f.close()
```

## What is JSON

- JSON stands for JavaScript Object Notation.
- JSON is lightweight data-interchange format.
- JSON is easy to read and write than XML.
- JSON is language independent.
- JSON supports array, object, string, number and values.
- JSON is a syntax for storing and exchanging data.

## JSON in Python

- Python has a built-in package called json, which can be use to work with JSON data.

### Example

```
import json
```

### Convert from JSON to Python

- If you have a JSON string you can parse it by using the json.loads() method.

### Example

#### Convert from JSON to Python:

```
import json
# some JSON:
x = '{ "name":"John", "age":30, "city":"New York"}'
# parse x:
y = json.loads(x)
# the result is a Python dictionary:
print(y["age"])
```

### Convert from Python to JSON

- If you have a Python object you can convert it into a JSON string by using the json.dumps() method.

#### Convert from Python to JSON:

```
import json
# a Python object (dict):
x = {
    "name": "John",
    "age": 30,
    "city": "New York"
}

# convert into JSON:
y = json.dumps(x)
# the result is a JSON string:
print(y)
```

You can convert Python objects of the following types, into JSON strings:

Dict , list , tuple , string , int , float , True , False , None

### Example

- Convert Python objects into JSON strings, and print the values:

```
import json
print(json.dumps({"name": "John", "age": 30}))
print(json.dumps(["apple", "bananas"]))
print(json.dumps(("apple", "bananas")))
print(json.dumps("hello"))
print(json.dumps(42))
print(json.dumps(31.76))
print(json.dumps(True))
print(json.dumps(False))
print(json.dumps(None))
```

Then you convert from Python to JSON, Python objects are converted into the JSON (JavaScript) equivalent:

| Python | JSON   |
|--------|--------|
| dict   | Object |
| list   | Array  |
| tuple  | Array  |
| str    | String |
| int    | Number |
| float  | Number |
| True   | true   |

### Example

#### Convert a Python object containing all the legal data types:

```
import json
x = {
    "name": "John",
    "age": 30,
    "married": True,
    "divorced": False,
    "children": ("Ann","Billy"),
    "pets": None,
    "cars": [
        {"model": "BMW 230", "mpg": 27.5},
        {"model": "Ford Edge", "mpg": 24.1}
    ]
}

print(json.dumps(x))
```



## Format the Result

- The example above prints a JSON string, but it is not very easy to read, with no indentations and line breaks.
- The `json.dumps()` method has parameters to make it easier to read the result:

### Example

- Use the `indent` parameter to define the numbers of indents:  
`json.dumps(x, indent=4)`
- You can also define the separators, default value is `(" ", ": ")`, which means using a comma and a space to separate each object, and a colon and a space to separate keys from values:

### Example

- Use the `separators` parameter change the default separator:  
`json.dumps(x, indent=4, separators=(". ", " = "))`

## Order the Result

- The `json.dumps()` method has parameters to order the keys in the result:

## Python and XML

- Python's original support for XML occurred with the release of version 1.5 and the `xml` module
- Python supports both document object model (DOM) tree-structured as well as event-based Simple API for XML (SAX) processing of XML documents
- SAX is a streaming interface, meaning that the documents are parsed and processed one line at a time via a continuous bytestream.
- This means that you can neither backtrack nor perform random access within an XML document

```
import xml.etree.ElementTree as xml
```

```
def createXML(filename):
```

```
    # Start with the root element
```

```
    root = xml.Element("users")
```

```
    children1 = xml.Element("user")
```

```
    root.append(children1)
```

```
    userId1 = xml.SubElement(children1, "id")
```

```
    userId1.text = "123"
```

```
    userName1 = xml.SubElement(children1, "name")
```

```
    userName1.text = "Jaydeep"
```

```
    tree = xml.ElementTree(root)
```

```
    with open(filename, "wb") as fh:
```

```
        tree.write(fh)
```

```
if __name__ == "__main__":
```

```
    createXML("test.xml")
```

## UNIT-5 Python and Data Analytics

- This chapter discusses the factors affecting the performance of machine learning models.
- The chapter provides technical definitions of performance for different types of machine learning problems.
- In an e-commerce application, for example, good performance might mean returning correct search results or presenting ads that site visitors frequently click.
- In a genetic problem, it might mean isolating a few genes responsible for a heritable condition. The chapter describes relevant performance measures for these different problems.
- The goal of selecting and fitting a predictive algorithm is to achieve the best possible performance.
- Achieving performance goals involves three factors:
  - 1) complexity of the problem,
  - 2) complexity of the algorithmic model employed
  - 3) the amount and richness of the data available.
- The chapter includes some visual examples that demonstrate the relationship between problem and model complexity and then provides technical guidelines for use in design and development.
- The Basic Problem: Understanding Function Approximation
- The problem statement for these problems has two types of variables:
  - 1) The variable that you are attempting to predict (for example, whether a visitor to a website will click an ad)
  - 2) Other variables (for example, the visitor's demographics or past behavior on the site) that you can use to make the prediction
- Problems of this type are referred to as **function approximation problems** because the goal is to construct a model generating predictions of the first of these as a function of the second.
- In a function approximation problem, the designer starts with a collection of historical examples for which the correct answer is known.
- For example, historical web log files will indicate whether a visitor clicked an ad when shown the ad.

- The data scientist next has to find other data that can be used to build a predictive model.
- For example, to predict whether a site visitor will click an ad, the data scientist might try using other pages that the visitor viewed before seeing the ad.
- If the user is registered with the site, data on past purchases or pages viewed might be available for making a prediction.
- The variable being predicted is referred to by a number of different names, such as target, label, and outcome.
- The variables being used to make the predictions are variously called predictors, regressors, features, and attributes.
- Determining what attributes to use for making predictions is called feature engineering.
- Data cleaning and feature engineering take 80 percent to 90 percent of a data scientist's time.

## Working with Training Data

### Explain Training Data in Data Analytics in Python

- The data scientist starts algorithm development with a training set.
- The training set consists of outcome examples and the assemblage of features chosen by the data scientist.
- The training set comprises two types of data:
  - 1) The outcomes you want to predict
  - 2) The features available for making the prediction
- an example of a training set.
- The leftmost column contains outcomes (whether a site visitor clicked a link) and features to be used to make predictions about whether visitors will click the link in the future.

#### Example Training Set

| OUTCOMES:<br>CLICKED ON<br>LINK | FEATURE1:<br>GENDER | FEATURE2:<br>MONEY SPENT<br>ON SITE | FEATURE3:<br>GENDER |
|---------------------------------|---------------------|-------------------------------------|---------------------|
| Yes                             | M                   | 0                                   | 25                  |
| No                              | F                   | 250                                 | 32                  |
| Yes                             | F                   | 12                                  | 17                  |

- The predictor values (a.k.a., features, attributes, and so on) can be arranged in the form of a matrix (see Equation 3-1).
- The notational convention used in this book is as follows. The table of predictors will be called  $X$ , and it has the following form:

$$X = \begin{matrix} & \begin{matrix} x_{11} & x_{12} & \dots & x_{1n} \end{matrix} \\ \begin{matrix} x_{21} \\ M \\ x_{m1} \end{matrix} & \begin{matrix} x_{22} \\ M \\ x_{m2} \end{matrix} & \begin{matrix} \dots \\ & \dots \end{matrix} & \begin{matrix} x_{2n} \\ O \\ x_{mn} \end{matrix} \end{matrix}$$

Notation for set of predictors

- Referring to the data set in Table 3.1,  $x_{11}$  would be M (gender),  $x_{12}$  would be 0.00 (money spent on site),  $x_{21}$  would be F (gender), and so on
- Sometimes it will be convenient to refer to all the attribute values for a particular example.
- For that purpose, (with a single index) will refer to the  $i$ th row of  $X$ .
- For the data set in Table 3.1,  $x_2$  would be a row vector containing the values F, 250, 32.
- Using the example of predicting ad clicks, the predictors might include demographic data about the site visitor like marital status and yearly income, etc...
- Attributes such as marital status, gender, or the state of residence go by several different designations. They may be called **factor** or **categorical**.
- Attributes like age or income that are represented by numbers are called **numeric** or **real-valued**
- The distinction between these two types of attributes is important because some algorithms may not handle one type or the other. For example, linear methods, including the ones covered in this book, require **numeric attributes**

- The targets corresponding to each row in  $X$  are arranged in a column vector  $Y$  as follows:

$$Y = \begin{pmatrix} y_1 \\ y_2 \\ \vdots \\ y_m \end{pmatrix}$$

#### Notation for vector of targets

- The target  $y_i$  corresponds to  $x_i$ —the predictors in the  $i$ th row of  $X$ .
- Targets may be of several different forms.
- For example, they may be real numbers, like if the objective were to predict how much a customer will spend.
- When the targets are real numbers, the problem is called a **regression problem**.
- If the targets are two-valued the problem is called a **binary classification problem**
- Predicting whether a customer will click an advertisement is a binary classification problem.
- If the targets contain several discrete values, the problem is a **multiclass classification problem**.
- Predicting which of several ads a customer will click would be a multiclass classification problem.

## Assessing Performance of Predictive Models

### Explain Assessing Performance of Predictive Models

- Good performance means using the attributes  $x_i$  to generate a prediction that is close to  $y_i$ , but close has different meanings for different problems.
- For a regression problem where  $y_i$  is a real number, performance is measured in terms like the **mean squared error (MSE)** or the **mean absolute error (MAE)**

$$\text{Mean squared error} = \left( \frac{1}{m} \right) \sum_{i=1}^m (y_i - \text{pred}(x_i))^2$$

$$\text{Mean absolute error} = \left( \frac{1}{m} \right) \sum_{i=1}^m |y_i - \text{pred}(x_i)|$$

- In a regression problem, the target ( $y_i$ ) and the prediction,  $\text{pred}(x_i)$ , are both real numbers, so it makes sense to describe the error as being the numeric difference between them.
- MSE squares the errors and averages over the data set to produce a measure of the overall level of errors
- MAE averages the absolute values of the errors instead of averaging the squares of the errors.

### **Factors Driving Algorithm Choices and Performance—Complexity and Data**

- Several factors affect the overall performance of a predictive algorithm.
- Among these factors are the complexity of the problem, the complexity of the model used, and the amount of training data available.
- The following sections describe how these factors interrelate to determine performance.

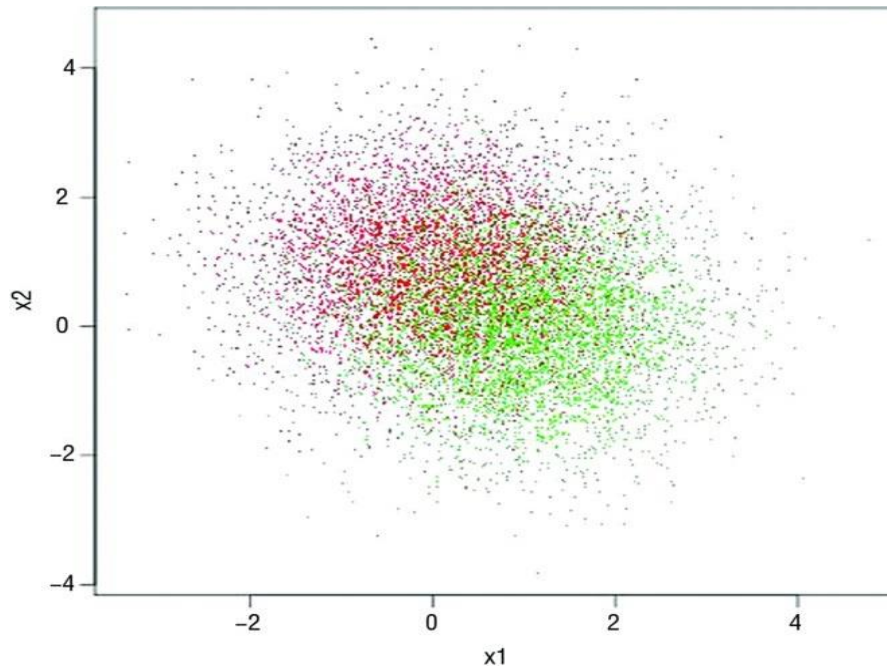
### **Contrast Between a Simple Problem and a Complex Problem**

### **Explain Simple Problem and Complex Problem in predictive model building in Python**

- The above section described several ways to quantify performance and highlighted the importance of performance on new data.
- The goal of designing a predictive model is to make accurate predictions on new examples (such as new visitors to your site).
- As a practicing data scientist, you will want an estimate of an algorithm's performance so that you can set expectations with your customer and compare algorithms with one another.

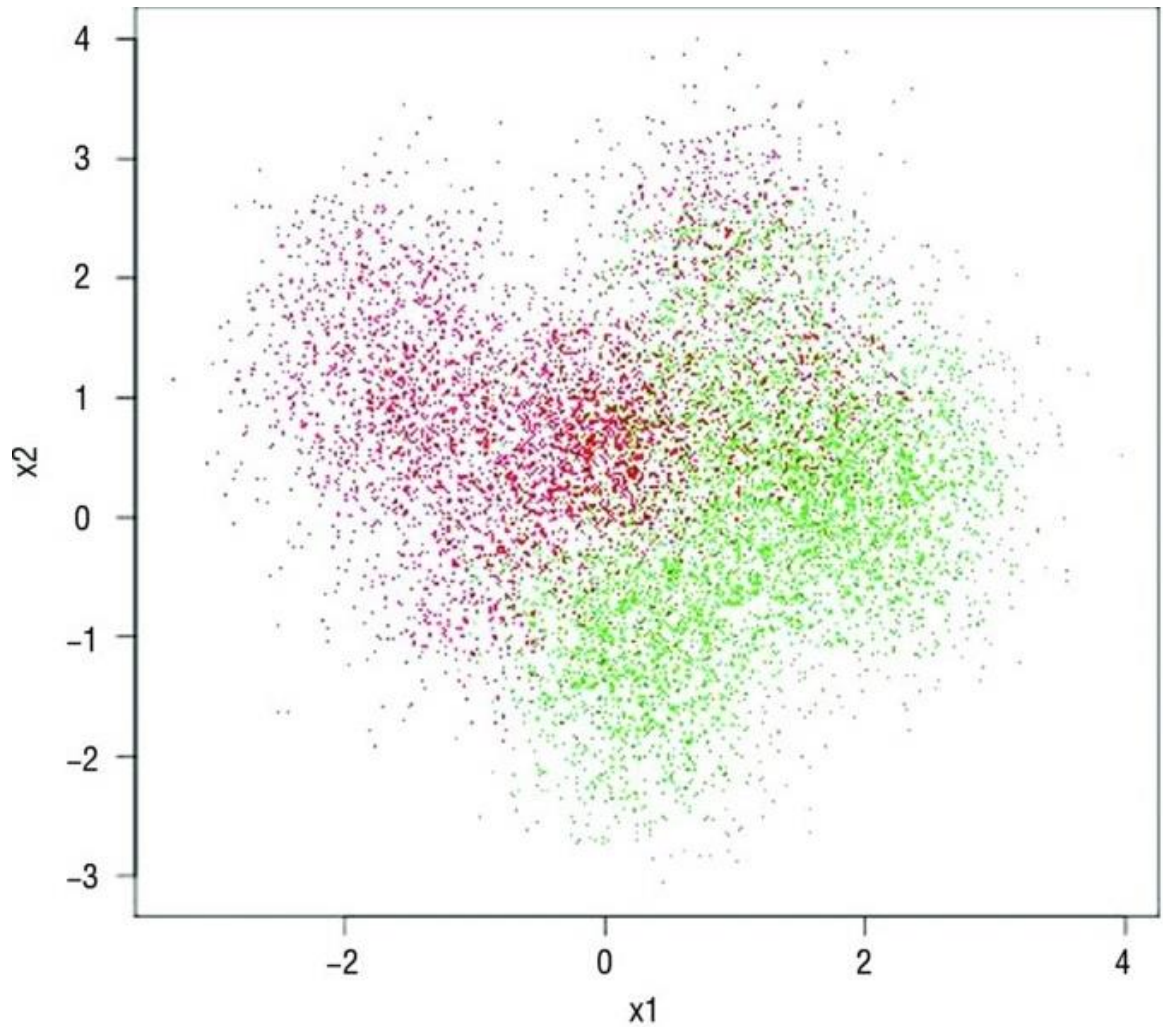
- Best practice in predictive modeling requires that you hold out some data from the training set.
- These held-out examples have labels associated with them and can be compared to predictions produced by models training on the remaining data.
- Statisticians refer to this technique as out-of-sample error because it is an error on data not used in training.
- The important thing is that the only performance that counts is the performance of the model when it is run against new examples.
- One of the factors affecting performance is the complexity of the problem being solved.
- Following Figure shows a relatively simple classification problem in two dimensions.
- There are two groups of points: dark and light points.
- The dark points are randomly drawn from a 2D Gaussian distribution centered at (1,0) with unit variance in both dimensions.
- The light points are also drawn from a Gaussian distribution having the same variance but centered at (0,1).
- The attributes for the problem are the two axes in the plot:  $X_1$  and  $X_2$ .
- The classification task is to draw some boundaries in the  $X_1, X_2$  plane to separate the light points from the dark points.
- About the best that can be done in this circumstance is to draw a 45-degree line in the plot—that is the line where  $X_1$  equals  $X_2$ .
- In a precise probabilistic sense, that is the best possible classifier for this problem.

### **A simple classification problem**



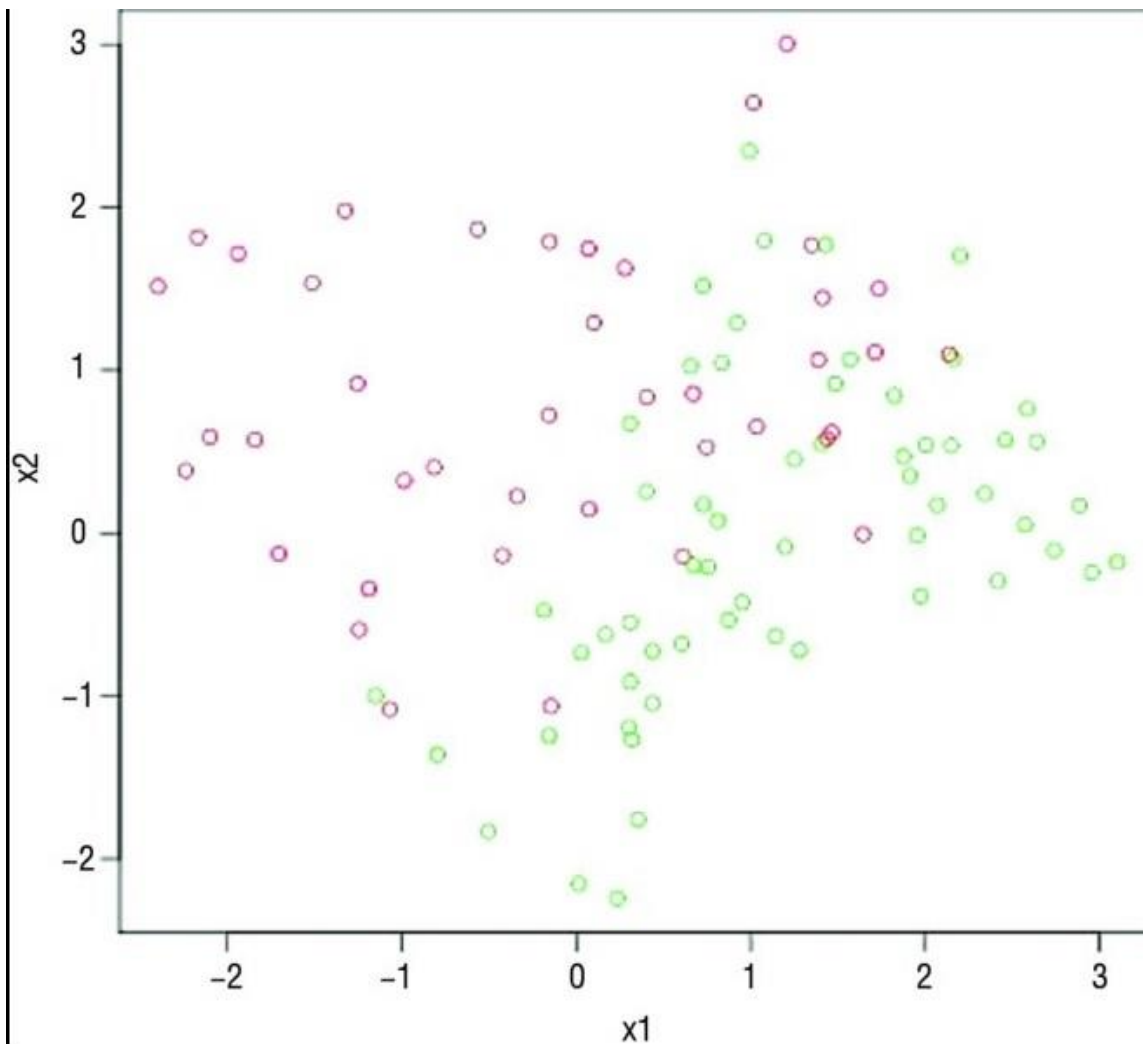
- Following Figure depicts a more complicated problem.
- The points shown in following Figure are generated by drawing points at random.
- The main difference from the random draw that generated above Figure is that the points in the below Figure are drawn from several distributions for the light points and several different ones for dark.
- This is called a mixture model. The general goal is basically the same:
- draw boundaries in the  $X_1$  and  $X_2$  plane to separate the light points from the dark points.
- In below Figure however, it is clear that a linear boundary will not separate the points as well as a curve.





### **A complicated classification problem**

- However, complexity of the decision boundaries is not the only factor influencing whether linear or nonlinear methods will deliver better performance.
- Another important factor is the size of the data set.
- Following Figure illustrates this element of performance.
- The points plotted in Figure are a 1 percent subsample of data plotted in above Figure



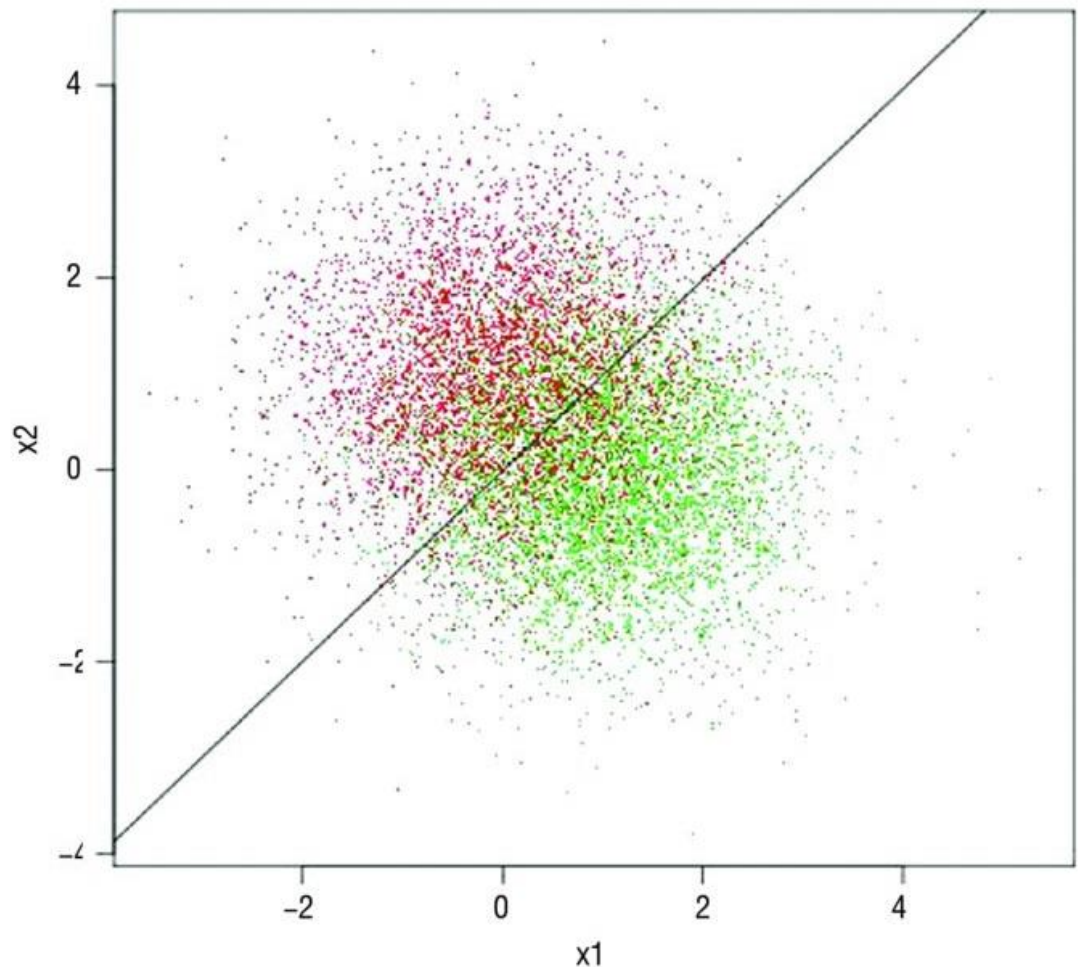
**A complicated classification problem without much data**

- In Figure 2 there was enough data to visualize the curved boundaries delineating the sets of light and dark points.
- Without as much data, the sets are not so easily discerned visually, and in this circumstance a linear model may give equal or better performance than a nonlinear model.
- However, if the model is not complicated, as in Figure 1, or there is not sufficient data, as in Figure 3, a linear model may produce the best answer.

### **Contrast Between a Simple Model and a Complex Model**

- The previous section showed visual comparisons between simple and complex problems.

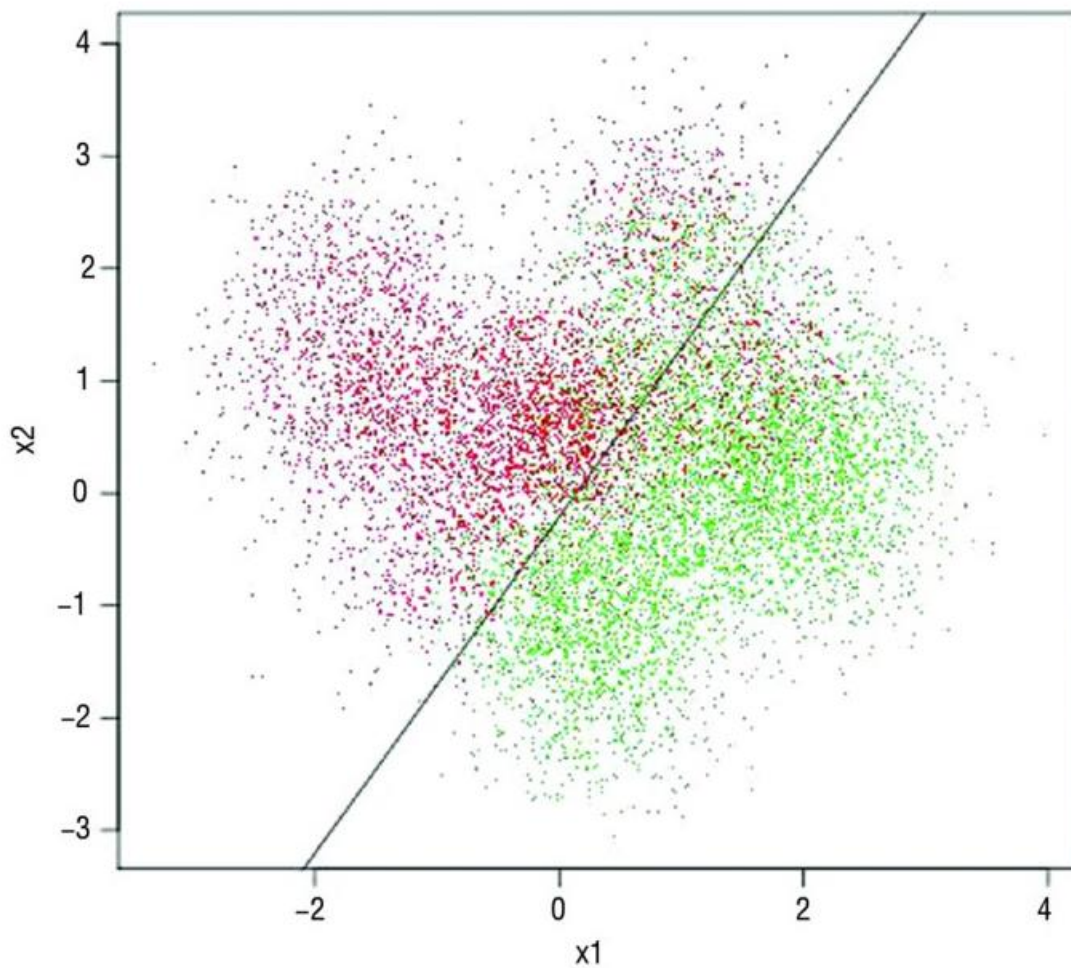
- This section describes how the various models available to solve these problems differ from one another.
- Intuitively (Naturally) it seems that a complex model should be fit to a complex problem, but the visual example from the last section demonstrates that data set size may dictate that a simple model fits a complex problem better than a complex model.



**Linear model fit to simple problem**

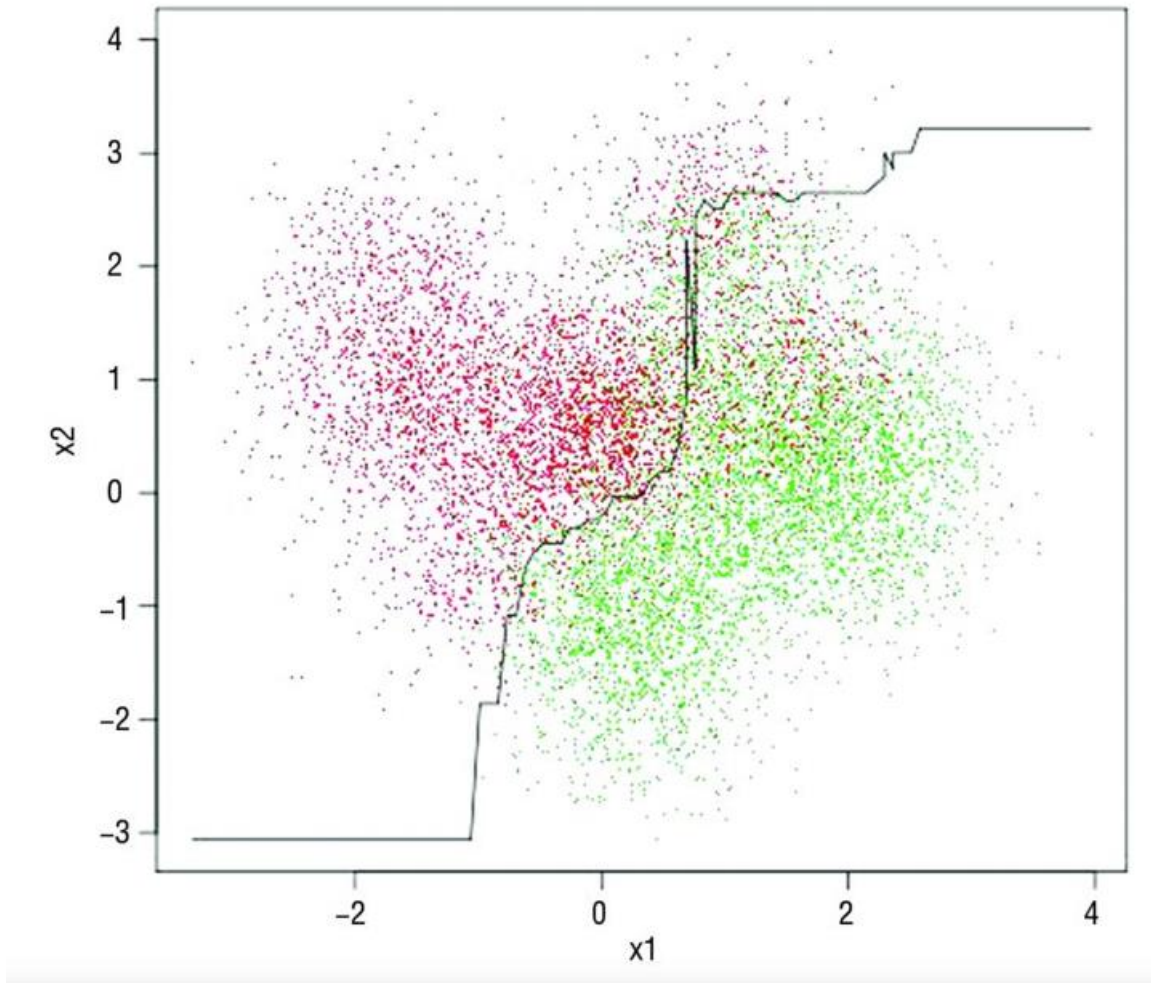
- Above Figure shows a linear model fit to the simple problem
- The model was generated using the **glmnet algorithm**
- The linear model fit to these data divides the data roughly in half. The line in the figure is given by Equation .  

$$x_2 = 0.01 + 0.99x_1$$
- A more complicated problem with more complicated decision boundaries gives a complicated model an opportunity to outperform a simple linear model



### Linear model fit to simple data

- Above Figure a linear model fit to data indicating a nonlinear decision boundary.
- In this circumstance, the linear model misclassifies regions as dark when they should be light and vice versa.
- Following Figure A shows how much better a complicated model can do with complicated data.
- The model used to generate this decision boundary is an ensemble (collection) of 1,000 binary decision trees constructed using the gradient boosting algorithm.
- The nonlinear decision boundary curves are used to better delineate regions where the dark points are denser and regions where the light points are denser.



**Ensemble model fit to complex data**

### **Factors Driving Predictive Algorithm Performance**

- These results explain the excitement over large volumes of data.
- Accurate predictions for complicated problems require large volumes of data.
- But the size isn't quite a precise enough measure.
- The shape of the data also matters.

### **Choosing an Algorithm: Linear or Nonlinear?**

- The visual examples you have just seen give some idea of the performance tradeoffs between linear and nonlinear predictive models.
- Linear models are preferable when the data set has more columns than rows or when the underlying problem is simple.

- Nonlinear models are preferable for complex problems with many more rows than columns of data.
- An additional factor is training time. Fast linear techniques train much faster than nonlinear techniques.
- Choosing a nonlinear model (say an ensemble method) entails training a number of different models of differing complexity.
- For example, the ensemble model that generated the decision boundary in Figure – A was one of roughly a thousand different models generated during the training process.
- These models had a variety of different complexities.
- Some of them would have given a much cruder approximation to the boundaries that are visually apparent in Figure - A.
- This section has used data sets and classifier solutions that can be visualized in order to give you an intuitive (in-built) grasp of the factors affecting the performance of the predictive models you build.
- Generally, you'll use numeric measures of performance instead of relying on pictures.
- The next section describes the methods and considerations for producing numeric performance measures for predictive models and how to use these to estimate the performance your models will achieve when deployed.

## Measuring the Performance of Predictive Models

- This section covers two broad areas relating to performance measures for predictive models.
- The first one is the different metrics that you can use for different types of problems (for example, using MSE for a regression problem and misclassification error for a classification problem).
- In the literature (and in machine learning competitions), you will also see measures like **receiver operating curves (ROC curves)** and **area under the curve (AUC)**.
- Besides that, these ideas are useful for optimizing performance.
- The second broad area consists of techniques for gathering out-of-sample error estimates.



- Recall that out-of-sample errors are meant to simulate errors on new data.
- It's an important part of design practice to use these techniques to compare different algorithms and to select the best model complexity for a given problem complexity and data set size.

## Performance Measures for Different Types of Problems

- Performance measures for regression problems are relatively straightforward.
- In a regression problem, both the target and the prediction are real numbers.
- Error is naturally defined as the difference between the target and the prediction.
- It is useful to generate statistical summaries of the errors for comparisons and for diagnostics.
- The most frequently used summaries are the **mean squared error (MSE)** and the **mean absolute error (MAE)**.

## Simulating Performance of Deployment Models

- The examples from the preceding section demonstrated the need for testing performance on data not included in the training set to get a useful estimate of expected performance once a predictive model is deployed
- The example broke the available labeled data into two subsets
- One subset called the training set contained approximately two-third of the available data and was used to fitting an ordinary least squares model
- The second subset which contained the remaining third of the available data was called the test set and was used only for determining performance (not used during training of the model)
- This is a standard procedure in machine learning
- Test set sizes range from 25 percent to 35 percent of the trained model deteriorates (fails) as the size of the training data set shrinks
- Taking out too much data from the training set can prove detrimental(harmful) to end performance

|                                |                |                |                |                |
|--------------------------------|----------------|----------------|----------------|----------------|
| Block 1                        | <b>Block 2</b> | <b>Block 3</b> | <b>Block 4</b> | <b>Block 5</b> |
| <b>Block 1</b>                 | Block 2        | <b>Block 3</b> | <b>Block 4</b> | <b>Block 5</b> |
| <b>N Fold Cross Validation</b> |                |                |                |                |

- Another approach to holding out data is called n-fold cross validation
- Above figure shows schematically how a data set is divided up for training and testing with n-fold cross validation
- The set is divided into n disjointed sets of roughly equal sizes
- In the figure n is 5
- Several training and testing passes are made through the data
- In the first pass the first block of data is held out for testing and the remaining n-1 are used for training
- In the second pass the second block is held out for testing and the other n-1 are used for training
- This process is continued until all the data have been held out( five times for the five fold example depicted in figure)
- The n-fold cross-validation process yields an estimate of the predication error and has several sampled of the erro so that it can estimate error bounds on the error
- It can keep more of the data in the training set which generally gives lower generalization errors and better final performnace
- For example if the 10-fold cross validation is chosen the only 10 percent of the data is held out for each training pass
- These features of n-fold cross validation come at the expenses of taking more training time
- The approch of taking a fixed holdout set has the advantage of faster training because it employs only on pass through the training data
- After a model has been trainned and tested it is a good practice to recombine the training and test data into a single set and retrain the model on the larger data set
- This section supplied you with tools to quantify the preformance of your predictive model



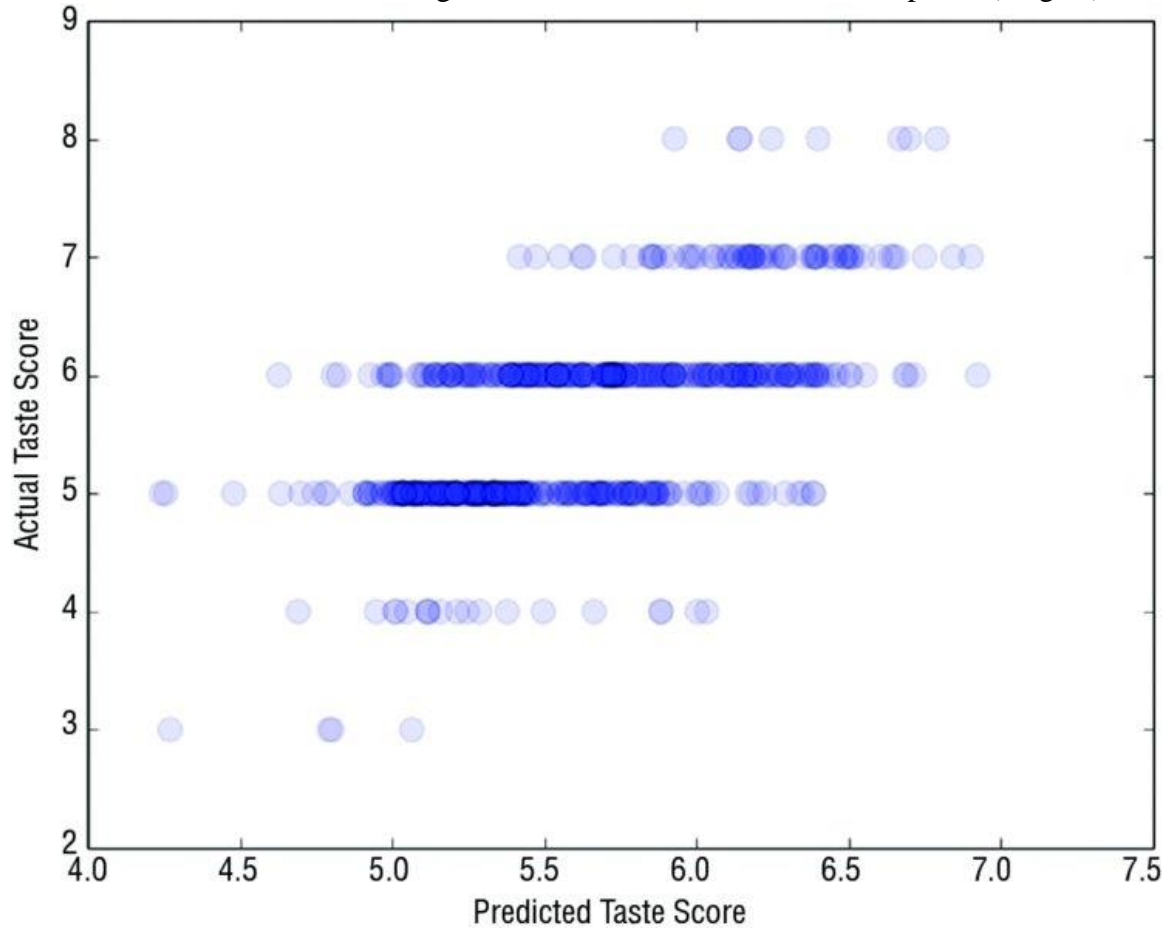
## Achieving Harmony Between Model and Data

- This section uses ordinary least squares (OLS) regression to illustrate several things.
- First, it illustrates how OLS can sometimes overfit a problem.
- Overfitting means that there's a significant discrepancy between errors on the training data and errors on the test data
- Second, it introduces two methods for overcoming the overfit problem with OLS. These methods will cultivate your intuition(perception)
- In addition, the methods for overcoming overfitting have a property that is common to most modern machine learning algorithms.
- Modern algorithms generate a number of models of varying complexity and then use out-of-sample performance to balance model complexity, problem complexity, and data set richness and thus determine which model to deploy.
- Fortunately, there's been a lot of work on ordinary least squares regression since its invention more than 200 years ago by Gauss and Legendre.
- This section introduces two of the methods for adjusting the throttle on ordinary least squares regression.
- One is called forward stepwise regression; the other is called **ridge regression**.

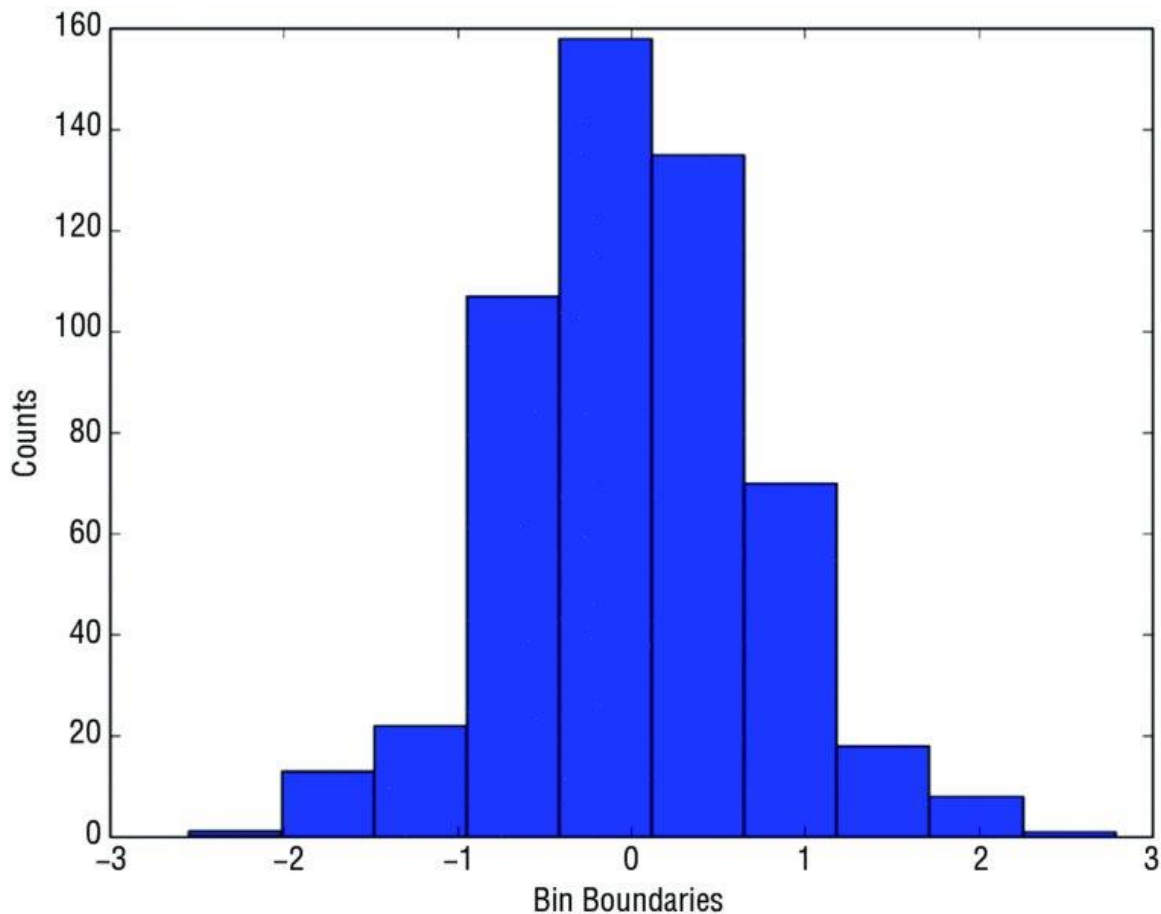
## Evaluating and Understanding Your Predictive Model

- Several other plots are helpful in understanding the performance of a trained algorithm and can point the way to making improvements in its performance.
- Figure A shows a scatter plot of the true labels plotted versus the predicted labels for points in the test set.
- the scatter plot shows horizontal rows of points.
- When the true values take on a small number of values, it is useful to make the data points partially transparent so that the darkness can indicate the accumulation of many points in one area of the graph.
- Actual taste scores of 5 and 6 are reproduced fairly well. The more extreme values are not as well predicted by the system.

- Figure B shows a histogram of the prediction error for forward stepwise prediction predicting wine taste scores.
- Sometimes the error histogram will have two or more discrete peaks (heights)



**Figure –A Actual taste scores versus predictions generated with forward stepwise regression**



**Figure -2 Histogram of wine taste prediction error with forward stepwise regression**

- Perhaps it will have a small peak on the far right or far left of the graph.
- In that case, it may be possible to find an explanation for the different peaks in the error and to reduce the prediction error by adding a new attribute that explains the membership in one or the other of the groups of points.
- You want to note several things about this output.
- First, let's reiterate the process.
- The process is to train a family of models (in this case, ordinary linear regression trained on column-wise subsets of X).
- The series of models is parameterized (in this case, by the number of attributes that are used in the linear model).

- The model to deploy is chosen to minimize the out-of-sample error. The number of attributes to be incorporated in the solution can be called a complexity parameter.
- Models with larger complexity parameters have more free parameters and are more likely to overfit the data than less-complex models.
- Also note that the attributes have become ordered by their importance in predicting quality.
- In the list of column numbers and the associated list of attribute names, the first in the list is the first attribute chosen, the second was next, and so on.
- The attributes used come out in a nice ordered list.
- This is an important and desirable feature of a machine learning technique. Early stages of a machine learning task mostly involve hunting for (or constructing) the best set of attributes for making predictions.
- Having techniques to rank attributes in order of importance helps in that process. The other algorithms developed in this book will also have this property.
- The last observation regards picking a model from the family that machine learning techniques generate.
- The more complicated the model, the less well it will generalize.
- It is better to error on the side of a less-complicated model.
- The earlier example indicates that there's very little degradation in performance between the 9th (best) model and the 10th model (a change in the 4th significant digit). Best practice would be to remove those attributes even if they were better in the 4th significant digit in order to be conservative.

### **What is predictive modeling ?**

- Predictive modeling is the general concept of building a model that is capable of making predictions.
- a model includes a machine learning algorithm that learns certain properties from a training dataset in order to make those predictions.
- Predictive modeling can be divided further into two sub areas:
  - 1) Regression
  - 2) pattern classification

- Regression models are based on the analysis of relationships between variables and trends in order to make predictions about continuous variables
- e.g. the prediction of the maximum temperature for the upcoming days in weather forecasting.
- In contrast to regression models, the task of pattern classification is to assign discrete class labels to particular observations as outcomes of a prediction.
- To go back to the above example: A pattern classification task in weather forecasting could be the prediction of a sunny, rainy, or snowy day.
- To not get lost in all possibilities, the main focus of this article will be on “pattern classification”, the general approach of assigning predefined class labels to particular instances in order to group them into discrete categories.
- The term “instance” is synonymous to “observation” or “sample” and describes an “object” that consists of one or multiple features (synonymous to “attributes”).