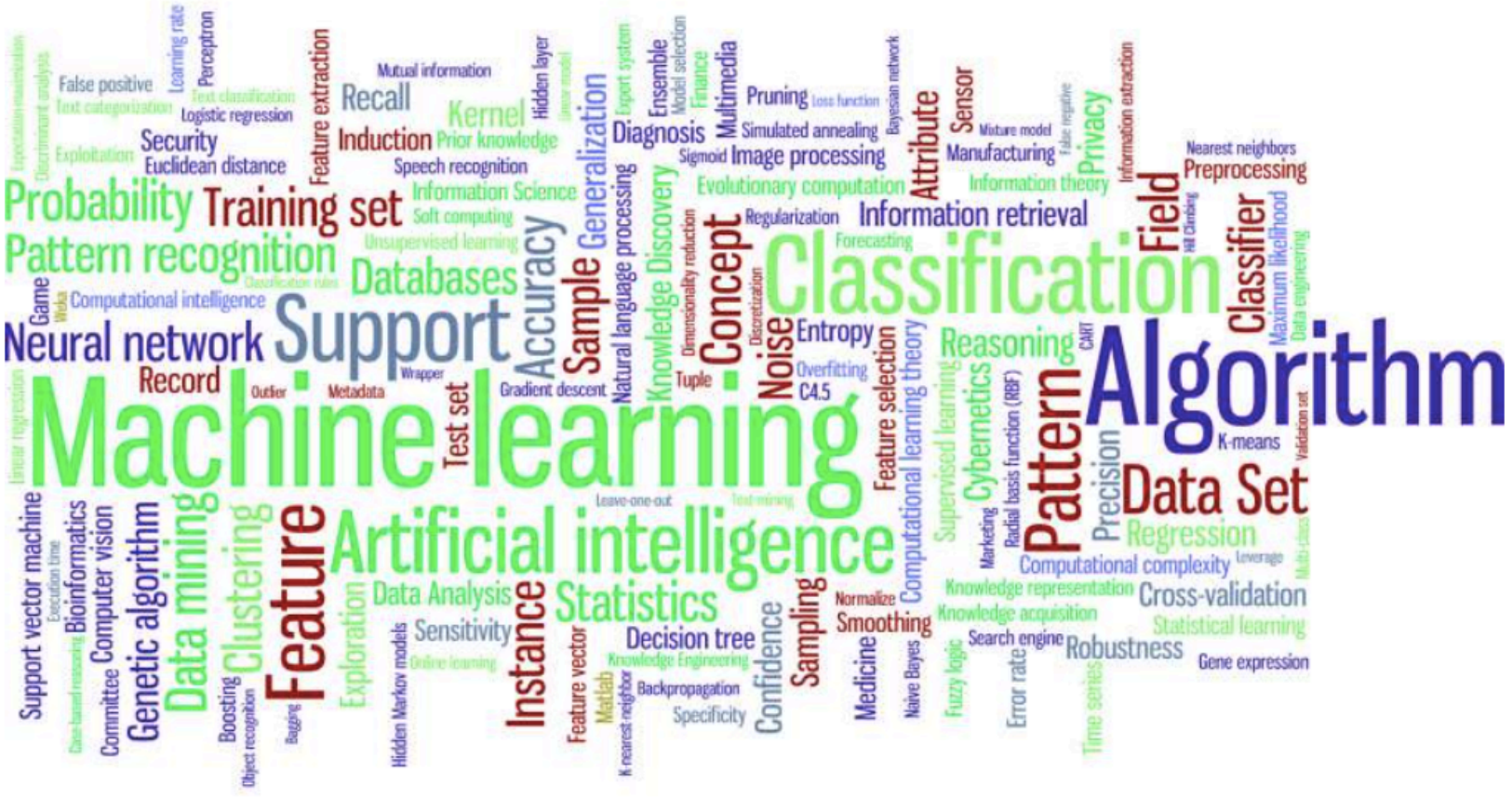
AI / ML COURSE


Machine learning
Algorithm
Artificial intelligence
Classification
Pattern recognition
Probability
Training set
Support
Neural network
Data Set
Feature
Statistics
Data mining
Clustering
Genetic algorithm
Support vector machine
Bioinformatics
Computer vision
Instance
Sampling
Confidence
Decision tree
Regression
Cross-validation
Robustness
Gene expression
Statistical learning
Computational complexity
Knowledge representation
Knowledge acquisition
Search engine
Cybernetics
Reasoning
Classifier
Field
Information retrieval
Information theory
Manufacturing
Sensor
Attribute
Privacy
Preprocessing
Nearest neighbors
Concept
Noise
Entropy
Overfitting
C4.5
Feature selection
Computational learning theory
Supervised learning
Precision
K-means
Accuracy
Sample
Generalization
Knowledge Discovery
Natural language processing
Databases
Induction
Recall
Kernel
Prior knowledge
Speech recognition
Information Science
Soft computing
Unsupervised learning
Diagnosis
Image processing
Evolutionary computation
Regularization
Forecasting
Pruning
Simulated annealing
Multimedia
Ensemble
Mutual information
Hidden layer
Feature extraction
False positive
Logistic regression
Perceptron
Security
Euclidean distance
Exploitation
Game
Computational intelligence
Record
Outlier
Metadata
Wrapper
Test set
Gradient descent
Tuple
Dimensionality reduction
Discretization
Exploration
Data Analysis
Sensitivity
Hidden Markov models
Boosting
Bagging
Committee
Genetic algorithm
Data mining
Clustering
Feature
Support vector machine
Bioinformatics
Computer vision
Instance
Feature vector
Matlab
K-nearest neighbor
Backpropagation
Specificity
Knowledge Engineering
Online learning
Decision tree
Confidence
Sampling
Normalize
Smoothing
Medicine
Naive Bayes
Fuzzy logic
Error rate
Time series
Robustness
Search engine
Gene expression
Statistical learning
Cross-validation
Knowledge acquisition
Knowledge representation
Computational complexity
Leverage
Marketing
Radial basis function (RBF)
Pattern
Data Set
Algorithm
Machine learning

PYTHON 3	MACHINE LEARNING	DEEP LEARNING	DATA SCIENCE
DATA ANALYSIS	NUMPY	PANDAS	MATPLOTLIB
SEABORN	TENSORFLOW	ANACONDA	JUPYTER
STATISTICS	PROBABILITY	NEURAL NETWORK	…….

Python3

Overview

- About
- History
- Installing & Running Python
- Names & Assignment
- Sequences types: Lists, Tuples, and Strings
- Mutability

ABOUT

- Python is a general-purpose interpreted, interactive, object-oriented, and high-level programming language.
- Open sourced from the beginning.
- Considered a scripting language, but is much more.
- Scalable and functional from the beginning.
- Increasingly popular.

Brief History of Python

- It was created by Guido van Rossum during 1985 – 1990 in Netherlands.
- Python 3 was released in 2008.
- Python is derived from many other languages, including ABC, Modula-3, C, C++, Algol-68, SmallTalk, and Unix shell and other scripting languages.

WHY PYTHON ?

- Python is designed to be highly readable. It uses English keywords frequently whereas the other languages use punctuations. It has fewer syntactical constructions than other languages.
- Python is Interpreted: Python is processed at runtime by the interpreter. You do not need to compile your program before executing it. This is similar to PERL and PHP.
- Python is Interactive: You can actually sit at a Python prompt and interact with the interpreter directly to write your programs.
- Python is Object-Oriented: Python supports Object-Oriented style or technique of programming that encapsulates code within objects.
- Python is a Beginner's Language: Python is a great language for the beginner- level programmers and supports the development of a wide range of applications from simple text processing to WWW browsers to games.

Features

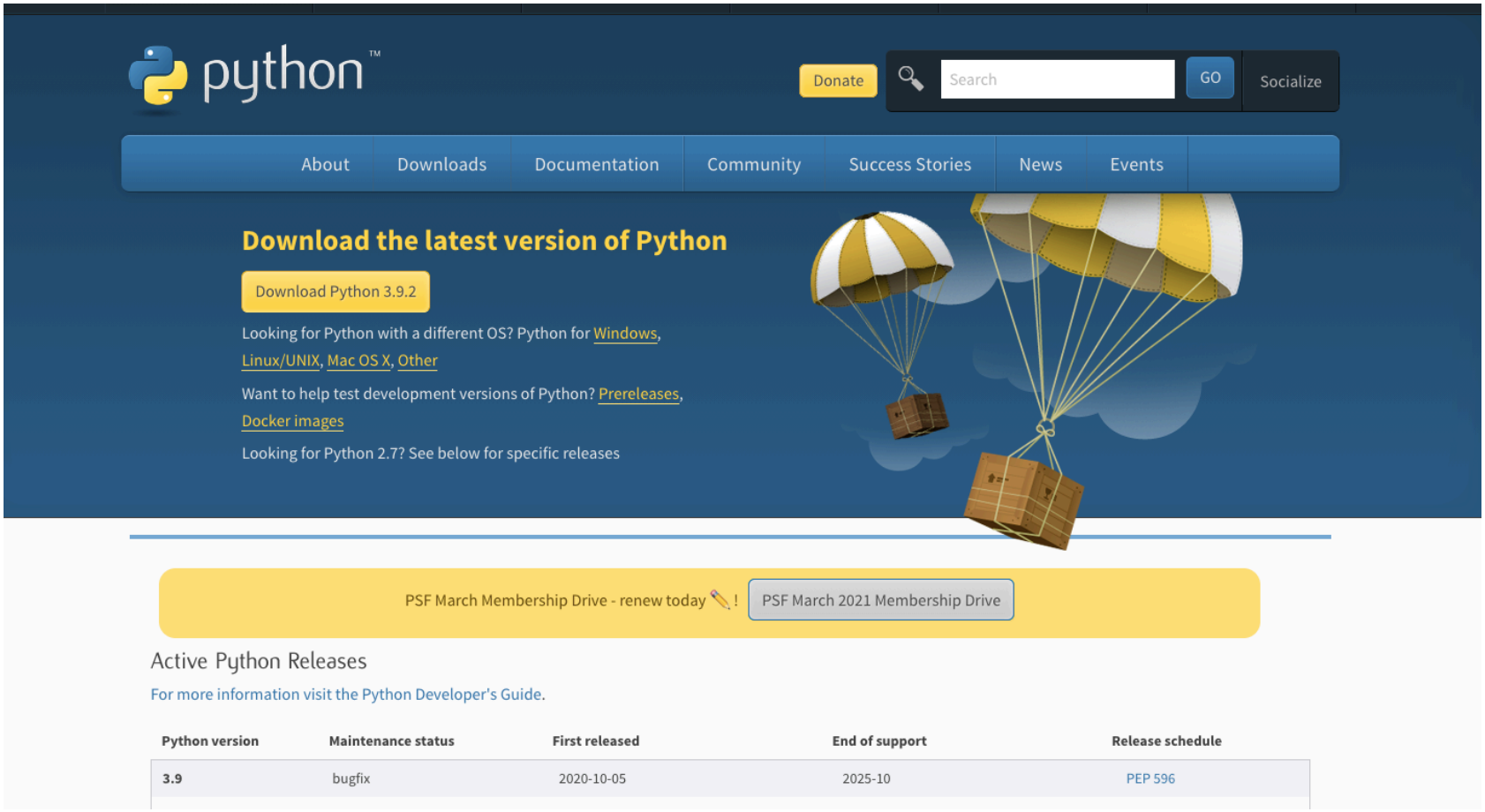
- Easy-to-learn
- Easy-to-read
- Easy-to-maintain
- A broad standard library
- Portable
- Extendable
- Databases
- GUI Programming
- Scalable

Installing

- Python is pre-installed on most Unix systems, including Linux and MAC OS X
- The pre-installed version may not be the most recent one (3.5 and 3.8 as of March 21).
- Download from http://python.org/download/
- Python comes with a large library of standard modules.
- There are several options for an IDE
  - IDLE – works well with Windows
  - Sublime Text Editor.
  - Anaconda, Jupyter Notebook.
  - Eclipse with Pydev (http://pydev.sourceforge.net/)

https://phoenixnap.com/kb/how-to-install-python-3-windows

https://www.python.org/downloads/



https://docs.anaconda.com/anaconda/install/

Installation

Review the system requirements listed below before installing Anaconda Individual Edition. If you don't want the hundreds of packages included with Anaconda, you can install Miniconda, a mini version of Anaconda that includes just conda, its dependencies, and Python.

Tip

Looking for Python 3.5 or 3.6? See our FAQ.

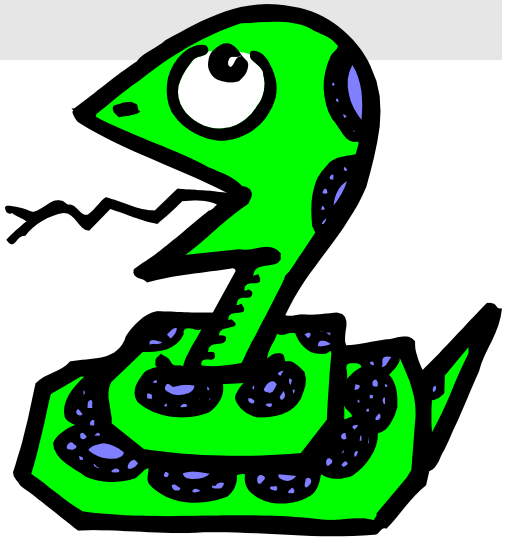
System requirements

- License: Free use and redistribution under the terms of the ../eula.
- Operating system: Windows 8 or newer, 64-bit macOS 10.13+, or Linux, including Ubuntu, RedHat, CentOS 6+, and others.
- If your operating system is older than what is currently supported, you can find older versions of the Anaconda installers in our archive that might work for you. See Using Anaconda on older operating systems for version recommendations.
- System architecture: Windows- 64-bit x86, 32-bit x86; MacOS- 64-bit x86; Linux- 64-bit x86, 64-bit Power8/Power9.
- Minimum 5 GB disk space to download and install.

On Windows, macOS, and Linux, it is best to install Anaconda for the local user, which does not require administrator permissions and is the most robust type of installation. However, if you need to, you can install Anaconda system wide, which does require administrator permissions.

- Installing on Windows
- Installing on macOS
- Installing on Linux
- Installing on Linux POWER

Running Python



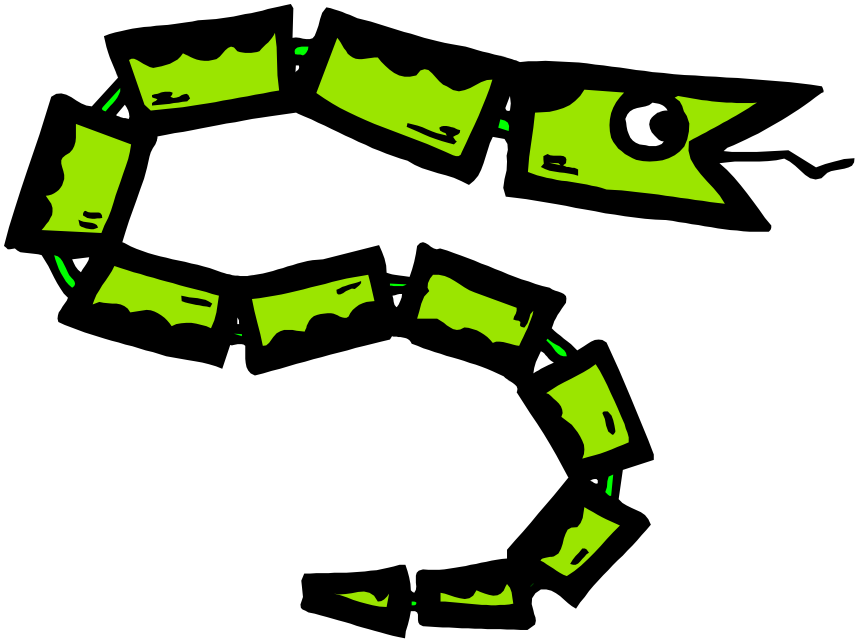
IDLE Development Environment

- IDLE is an Integrated Development Environment for Python, typically used on Windows.
- Multi-window text editor with syntax highlighting, auto-completion, smart indent and other.
- Python shell with syntax highlighting.
- Integrated debugger
  with stepping, persistent breakpoints,
  and call stack visibility

Python Scripts

- When you call a python program from the command line the interpreter evaluates each expression in the file
- Familiar mechanisms are used to provide command line arguments and/or redirect input and output
- Python also has mechanisms to allow a python program to act both as a script and as a module to be imported and used by another python program

The Basics



- print ("Hello World")
- The print() function inserts a new line at the end, by default. “ end=“ “ " appends space instead of newline.
- print(x, end=" ") # Appends a space instead of a newline.
- x=input("something:")
- x=input("")
  x=int(input(“”))

Assigning Values to Variables

```
counter = 100        # An integer assignment
 miles   = 1000.0    # A floating point
 name    = “John"    # A string
 print (counter)
 print (miles)
 print (name)

a=b=c=1
 a, b, c = 1, 2, "john"
```

COMMENT

- # This is a comment.
- # This is a comment, too.
- # This is a comment, too.
- # I said that already.

Naming Rules

- Names are case sensitive and cannot start with a number. They can contain letters, numbers, and underscores.

```
bob   Bob   _bob   _2_bob_
bob_2   BoB
```

- There are some reserved words:

```
and, assert, break, class, continue,
def, del, elif, else, except, exec,
finally, for, from, global, if,
import, in, is, lambda, not, or,
pass, print, raise, return, try,
while
```

Naming conventions

The Python community has these recommend-ed naming conventions

- joined_lower for functions, methods and, attributes
- joined_lower or ALL_CAPS for constants
- StudlyCaps for classes
- camelCase only to conform to pre-existing conventions
- Attributes: interface, _internal, __private

RESERVED WORDS

and	exec	Not
as	finally	or
assert	for	pass
break	from	print
class	global	raise
continue	if	return
def	import	try
del	in	while
elif	is	with
else	lambda	yield
except		

Standard Data Types

The data stored in memory can be of many types.

For example, a person's Weight is stored as a numeric value and his or her address is stored as alphanumeric characters.

Python has various standard data types that are used to define the operations possible on them and the storage method for each of them.

Python has five standard data types-

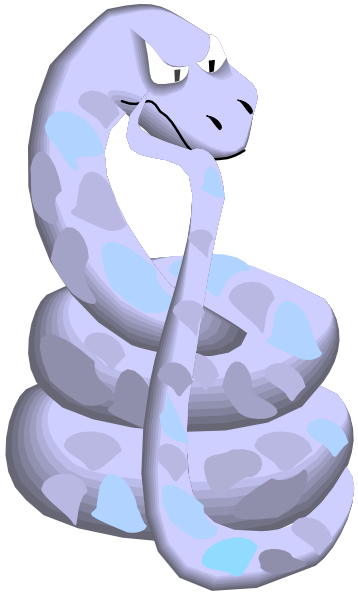
- Numbers
- String
- List
- Tuple
- Dictionary

```
var1 = 1
var2 = 10
```

```
del var1
del var1, var2
```

Python supports three different numerical types –

- int (signed integers)
- float (floating point real values)
- complex (complex numbers)



PYTHON STRINGS

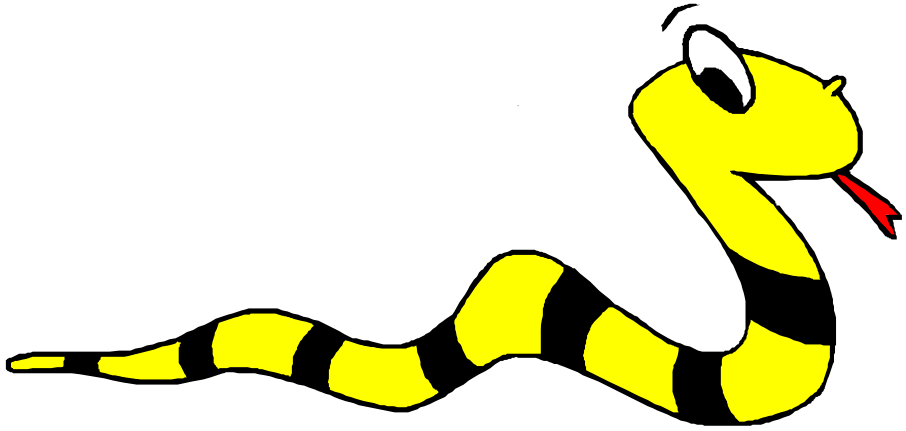
Strings in Python are identified as a contiguous set of characters represented in the quotation marks. Python allows either pair of single or double quotes. Subsets of strings can be taken using the slice operator ([ ] and [:]) with indexes starting at 0 in the beginning of the string and working their way from -1 to the end.

The plus (+) sign is the string concatenation operator and the asterisk (*) is the repetition operator. For example-

```
str = 'Hello World!'

print (str)      # Prints complete string

print (str[0]) # Prints first character of the string

print (str[2:5]) # Prints characters starting from 3rd to 5th

print (str[2:])  # Prints string starting from 3rd character

print (str * 2) # Prints string two times

print (str + "TEST") # Prints concatenated string
```

RESULTS

```
Hello World!
H
llo
llo World!
Hello World!Hello World!
Hello World!
```



Python Lists

Lists are the most versatile of Python's compound data types. A list contains items separated by commas and enclosed within square brackets ([]). To some extent, lists are similar to arrays in C. One of the differences between them is that all the items belonging to a list can be of different data type.

The values stored in a list can be accessed using the slice operator ([ ] and [:]) with indexes starting at 0 in the beginning of the list and working their way to end -1. The plus (+) sign is the list concatenation operator, and the asterisk (*) is the repetition operator. For example-

```
list = [ 'abcd', 786 , 2.23, 'john', 70.2 ]
tinylist = [123, 'john']
print (list) # Prints complete list
print (list[0]) # Prints first element of the list
print (list[1:3]) # Prints elements starting from 2nd till 3rd
print (list[2:]) # Prints elements starting from 3rd element
print (tinylist * 2) # Prints list two times
print (list + tinylist) # Prints concatenated lists
```

RESULTS

```
['abcd', 786, 2.23, 'john', 70.200000000000003] abcd
[786, 2.23]
[2.23, 'john', 70.200000000000003]

[123, 'john', 123, 'john']
['abcd', 786, 2.23, 'john', 70.200000000000003, 123, 'john']
```

Python Tuples

A tuple is another sequence data type that is similar to the list. A tuple consists of a number of values separated by commas. Unlike lists, however, tuples are enclosed within parenthesis.

The main difference between lists and tuples is- Lists are enclosed in brackets ([ ]) and their elements and size can be changed, while tuples are enclosed in parentheses (( )) and cannot be updated. Tuples can be thought of as read-only lists. For example-

```
tuple = ( 'abcd', 786 , 2.23, 'john', 70.2 )

tinytuple = (123, 'john')

print (tuple) # Prints complete tuple

print (tuple[0]) # Prints first element of the tuple

print (tuple[1:3]) # Prints elements starting from 2nd till 3rd

print (tuple[2:]) # Prints elements starting from 3rd element

print (tinytuple * 2) # Prints tuple two times

print (tuple + tinytuple) # Prints concatenated tuple
```

RESULTS

```
('abcd', 786, 2.23, 'john', 70.200000000000003)
abcd
(786, 2.23)
(2.23, 'john', 70.200000000000003)
(123, 'john', 123, 'john')
('abcd', 786, 2.23, 'john', 70.200000000000003, 123, 'john')
```

```
tuple = ( 'abcd', 786 , 2.23, 'john', 70.2 )

list = [ 'abcd', 786 , 2.23, 'john', 70.2 ]

tuple[2] = 1000 # Invalid syntax with tuple

list[2] = 1000 # Valid syntax with list
```

Python Dictionary

Python's dictionaries are kind of hash-table type. They work like associative arrays or hashes found in Perl and consist of key-value pairs. A dictionary key can be almost any Python type, but are usually numbers or strings. Values, on the other hand, can be any arbitrary Python object.

Dictionaries are enclosed by curly braces ({ }) and values can be assigned and accessed using square braces ([]). For example-

```
dict = {}
 dict['one'] = "This is one"
 dict[2]     = "This is two"
 tinydict = {'name': 'john','code':6734, 'dept': 'sales'}

print (dict[‘one']) # Prints value for 'one' key
print (dict[2]) # Prints value for 2 key

print (tinydict) # Prints complete dictionary

print (tinydict.keys()) # Prints all the keys

print (tinydict.values()) # Prints all the values
```

» RESULTS

```
This is one
This is two
{'dept': 'sales', 'code': 6734, 'name': 'john'}
['dept', 'code', 'name']
['sales', 6734, 'john']
```

»

CONVERSION OF DATA TYPES

int(x [,base]) Converts x to an integer. The base specifies the base if x is a string.

float(x) Converts x to a floating-point number.

complex(real [,imag]) Creates a complex number

str(x) Converts object x to a string representation.

repr(x) Converts object x to an expression string.

eval(str) Evaluates a string and returns an object.

tuple(s) Converts s to a tuple.

list(s) Converts s to a list.
set(s) Converts s to a set.
dict(d) Creates a dictionary. d must be a sequence of (key,value) tuples.

frozenset(s) Converts s to a frozen set.
chr(x) Converts an integer to a character.
unichr(x) Converts an integer to a Unicode character.
ord(x) Converts a single character to its integer value.
hex(x) Converts an integer to a hexadecimal string.
oct(x) Converts an integer to an octal string.

BASIC OPERATORS

Operators are the constructs, which can manipulate the value of operands.

Consider the expression 4 + 5 = 9.

Here, 4 and 5 are called operands and + is called the operator.

Types of Operator

Python language supports the following types of operators-

- Arithmetic Operators
- Comparison (Relational) Operators
- Assignment Operators
- Logical Operators
- Bitwise Operators
- Membership Operators
- Identity Operators

Let us have a look at all the operators one by one.

Assume variable **a** holds the value 10 and variable **b** holds the value 21, then-

Operator	Description	Example
+ Addition	Adds values on either side of the operator.	a + b = 31
- Subtraction	Subtracts right hand operand from left hand operand.	a – b = -11
* Multiplication	Multiplies values on either side of the operator	a * b = 210
/ Division	Divides left hand operand by right hand operand	b / a = 2.1
% Modulus	Divides left hand operand by right hand operand and returns remainder	b % a = 1
** Exponent	Performs exponential (power) calculation on operators	a**b =10 to the power 20
//	Floor Division - The division of operands where the result is the quotient in which the digits after the decimal point are removed.	9//2 = 4 and 9.0//2.0 = 4.0

» EXAMPLE

```
a = 21
b = 10
c=0

c=a+b
print ("Line 1 - Value of c is ", c)

c=a-b
print ("Line 2 - Value of c is ", c )

c=a*b
print ("Line 3 - Value of c is ", c)

c=a/b
print ("Line 4 - Value of c is ", c )

c=a%b
print ("Line 5 - Value of c is ", c)

a=2
b=3
c = a**b
print ("Line 6 - Value of c is ", c)

a = 10
b=5
c = a//b
print ("Line 7 - Value of c is ", c)
```

»

» OUTPUT

Line1-Valueofcis 31

Line2-Valueofcis 11

Line3-Valueofcis 210

Line4-Valueofcis 2.1

Line5-Valueofcis 1

Line6-Valueofcis 8

Line7-Valueofcis 2

»

Comparison Operators

These operators compare the values on either side of them and decide the relation among them. They are also called Relational operators.

Assume variable a holds the value 10 and variable b holds the value 20, then-

»

Operator	Description	Example
==	If the values of two operands are equal, then the condition becomes true.	(a == b) is not true.
!=	If values of two operands are not equal, then condition becomes true.	(a!= b) is true.
>	If the value of left operand is greater than the value of right operand, then condition becomes true.	(a > b) is not true.
<	If the value of left operand is less than the value of right operand, then condition becomes true.	(a < b) is true.
>=	If the value of left operand is greater than or equal to the value of right operand, then condition becomes true.	(a >= b) is not true.
<=	If the value of left operand is less than or equal to the value of right operand, then condition becomes true.	(a <= b) is true.

```
if ( a != b ):
   print ("Line 2 - a is not equal to b")
 else:
   print ("Line 2 - a is equal to b")
 if ( a < b ):
   print ("Line 3 - a is less than b" )
 else:
   print ("Line 3 - a is not less than b")
 if ( a > b ):
   print ("Line 4 - a is greater than b")
 else:
   print ("Line 4 - a is not greater than b")
a,b=b,a #values of a and b swapped. a becomes 10, b becomes 21

 if ( a <= b ):
   print ("Line 5 - a is either less than or equal to  b")
 else:
   print ("Line 5 - a is neither less than nor equal to  b")
»
```

OUTPUT

Line 2 - a is not equal to b
Line 3 - a is not less than b
Line 4 - a is greater than b
Line 5 - a is either less than or equal to b Line 6 - b is either greater than or equal to b

»

Assignment Operators

Assume variable a holds 10 and variable b holds 20, then-

»

Operator	Description	Example
=	Assigns values from right side operands to left side operand	c = a + b assigns value of a + b into c
+= Add AND	It adds right operand to the left operand and assign the result to left operand	c += a is equivalent to c = c + a

-= Subtract AND	It subtracts right operand from the left operand and assign the result to left operand	c -= a is equivalent to c = c - a
*= Multiply AND	It multiplies right operand with the left operand and assign the result to left operand	c *= a is equivalent to c = c * a
/= Divide AND	It divides left operand with the right operand and assign the result to left operand	c /= a is equivalent to c = c / ac /= a is equivalent to c = c / a
%= Modulus AND	It takes modulus using two operands and assign the result to left operand	c %= a is equivalent to c = c % a
**= Exponent AND	Performs exponential (power) calculation on operators and assign value to the left operand	c **= a is equivalent to c = c ** a
//= Floor Division	It performs floor division on operators and assign value to the left operand	c //= a is equivalent to c = c // a

Example

Assume variable a holds 10 and variable b holds 20, then-

```
a = 21 b = 10 c=0

c=a+b
print ("Line 1 - Value of c is ", c)

 c += a
 print ("Line 2 - Value of c is ", c )
 c *= a
 print ("Line 3 - Value of c is ", c )
 c /= a
 print ("Line 4 - Value of c is ", c )
c=2
c %= a
print ("Line 5 - Value of c is ", c)

 c **= a
 print ("Line 6 - Value of c is ", c)
 c //= a
 print ("Line 7 - Value of c is ", c)
»
```

» OUTPUT

Line1-Valueofcis 31

Line2-Valueofcis 52

Line 3 - Value of c is 1092

Line 4 - Value of c is 52.0

Line5-Valueofcis 2
Line 6 - Value of c is 2097152

Line 7 - Value of c is 99864

»

Bitwise Operators

Bitwise operator works on bits and performs bit-by-bit operation. Assume if a = 60; and b = 13; Now in binary format they will be as follows-

a = 0011 1100
b = 0000 1101

---

a&b = 0000 1100

a|b = 0011 1101

a^b = 0011 0001

~a = 1100 0011

Python's built-in function bin() can be used to obtain binary representation of an integer number.

»

The following Bitwise operators are supported by Python language-

Operator	Description	Example
& Binary AND	Operator copies a bit to the result, if it exists in both operands	(a & b) (means 0000 1100)
| Binary OR	It copies a bit, if it exists in either operand.	(a | b) = 61 (means 0011 1101)
^ Binary XOR	It copies the bit, if it is set in one operand but not both.	(a ^ b) = 49 (means 0011 0001)
~ Binary Ones Complement	It is unary and has the effect of 'flipping' bits.	(~a) = -61 (means 1100 0011 in 2's complement form due to a signed binary number.
<< Binary Left Shift	The left operand's value is moved left by the number of bits specified by the right operand.	a << = 240 (means 1111 0000)
>> Binary Right Shift	The left operand's value is moved right by the number of bits specified by the right operand.	a >> = 15 (means 0000 1111)

Logical Operators

The following logical operators are supported by Python language. Assume variable a holds True and variable b holds False then-

»

Operator	Description	Example
and Logical AND	If both the operands are true then condition becomes true.	(a and b) is False.
or Logical OR	If any of the two operands are non-zero then condition becomes true.	(a or b) is True.
not Logical NOT	Used to reverse the logical state of its operand.	Not(a and b) is True.

Membership Operators

Python’s membership operators test for membership in a sequence, such as strings, lists, or tuples. There are two membership operators as explained below-

»

Operator	Description	Example
in	Evaluates to true, if it finds a variable in the specified sequence and false otherwise.	x in y, here in results in a 1 if x is a member of sequence y.
not in	Evaluates to true, if it does not find a variable in the specified sequence and false otherwise.	x not in y, here not in results in a 1 if x is not a member of sequence y.

» EXAMPLE

```
a = 10
b = 20
list = [1, 2, 3, 4, 5 ]
if ( a in list ):
print ("Line 1 - a is available in the given list")

else:
   print ("Line 1 - a is not available in the given list")
if ( b not in list ):
   print ("Line 2 - b is not available in the given list")
else:
   print ("Line 2 - b is available in the given list")
c=b/a
if ( c in list ):

   print ("Line 3 - a is available in the given list")
else:
print ("Line 3 - a is not available in the given list”)
```

```
Line 1 - a is not available in the given list
Line 2 - b is not available in the given list
Line 3 - a is available in the given list
```

Operators Precedence

The following table lists all the operators from highest precedence to the lowest.

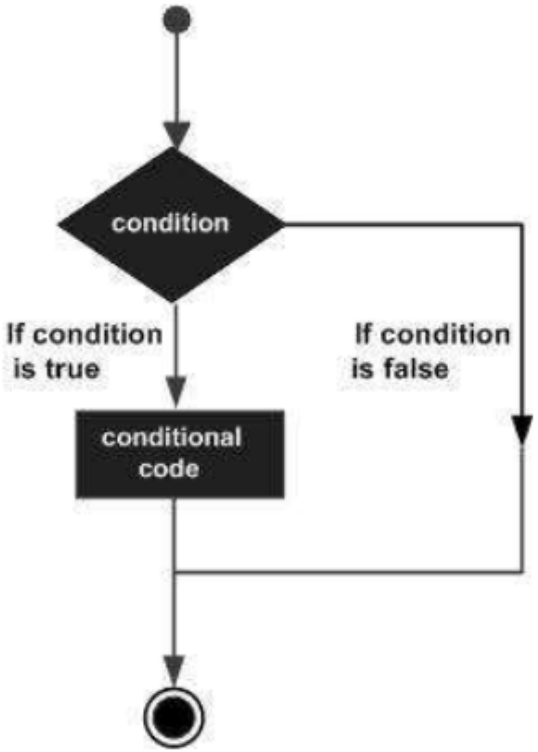
»

Operator	Description
**	Exponentiation (raise to the power)
~ + -	Ccomplement, unary plus and minus (method names for the last two are +@ and -@)
* / % //	Multiply, divide, modulo and floor division
+ -	Addition and subtraction
>> <<	Right and left bitwise shift
&	Bitwise 'AND'
^ |	Bitwise exclusive `OR' and regular `OR'
<= < > >=	Comparison operators
<> == !=	Equality operators

DecisionMaking

Decision structures evaluate multiple expressions, which produce TRUE or FALSE as the outcome. You need to determine which action to take and which statements to execute if the outcome is TRUE or FALSE otherwise.

»



if statements - An if statement consists of a Boolean expression followed by one or more statements.

if...else statements - An if statement can be followed by an optional else statement, which executes when the boolean expression is FALSE.

nested if statements - You can use one if or else if statement inside another if or else if statement(s).

EXAMPLE

```
var1 = 100
 if var1:
    print ("1 - Got a true expression value")

    print (var1)

 var2 = 0
 if var2:
    print ("2 - Got a true expression value")

    print (var2)

 print ("Good bye!")
```

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

```
amount=int(input("Enter amount: "))

if amount<1000:

     discount=amount*0.05
     print ("Discount",discount)
 else:
     discount=amount*0.10
     print ("Discount",discount)
print ("Net payable:",amount-discount)
```

» OUTPUT

```
Enter amount: 600
Discount 30.0
Net payable: 570.0
Enter amount: 1200
Discount 120.0
Net payable: 1080.0
»
```

```
if expression1:
   statement(s)
elif expression2:
   statement(s)
elif expression3:
   statement(s)
else:
   statement(s)
```

» EXAMPLE

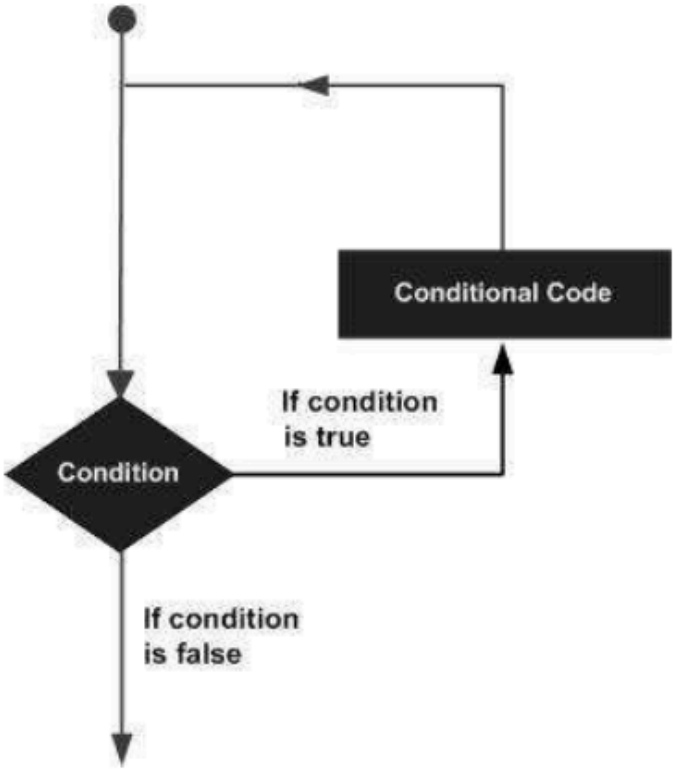
```
amount=int(input("Enter amount: "))

if amount<1000:
    discount=amount*0.05
    print ("Discount",discount)
elif amount<5000:
    discount=amount*0.10
    print ("Discount",discount)
else:
    discount=amount*0.15
    print ("Discount",discount)
print ("Net payable:",amount-discount)
```

»

Loops

A loop statement allows us to execute a statement or group of statements multiple times. The following diagram illustrates a loop statement.

»



while loop - Repeats a statement or group of statements while a given condition is TRUE. It tests the condition before executing the loop body.

for loop - Executes a sequence of statements multiple times and abbreviates the code that manages the loop variable.

nested loops - You can use one or more loop inside any another while, or for loop.

EXAMPLE

#WHILE LOOP

while expression:

    statement(s)

```
count = 0
while (count < 9):
   print ('The count is:', count)
   count = count + 1
print ("Good bye!")
```

The Infinite Loop

A loop becomes infinite loop if a condition never becomes FALSE. You must be cautious when using while loops because of the possibility that this condition never resolves to a FALSE value. This results in a loop that never ends. Such a loop is called an infinite loop.

#FOR LOOP

```
for iterating_var in sequence:

    statements(s)
```

```
for var in list(range(5)):
     print (var)
```

```
for letter in 'Python':
    print ('Current Letter :', letter)

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

for index in range(len(fruits)):

    print ('Current fruit :', fruits[index])
```

»

LOOP CONTROL STATEMENTS

break statement - Terminates the loop statement and transfers execution to the statement immediately following the loop.

continue statement - Causes the loop to skip the remainder of its body and immediately retest its condition prior to reiterating.

pass statement - The pass statement in Python is used when a statement is required syntactically but you do not want any command or code to execute.

»

Sequence types: Tuples, Lists, and Strings

Sequence Types

1. Tuple: (‘john’, 32, [CMSC])
   - A simple *immutable* ordered sequence of items
   - Items can be of mixed types, including collection types
2. Strings: “John Smith”
   - *Immutable*
   - Conceptually very much like a tuple
3. List: [1, 2, ‘john’, (‘up’, ‘down’)]
   - *Mutable* ordered sequence of items of mixed types

Similar Syntax

- All three sequence types (tuples, strings, and lists) share much of the same syntax and functionality.
- Key difference:
  - Tuples and strings are *immutable*
  - Lists are *mutable*
- The operations shown in this section can be applied to *all* sequence types
  - most examples will just show the operation performed on one

Sequence Types 1

- Define tuples using parentheses and commas
```
>>> tu = (23, 'abc', 4.56, (2,3), 'def')
```
- Define lists are using square brackets and commas
```
>>> li = ["abc", 34, 4.34, 23]
```
- Define strings using quotes (", ', or """).
```
>>> st = "Hello World"
>>> st = 'Hello World'
>>> st = """This is a multi-line
string that uses triple quotes."""
```

Sequence Types 2

- Access individual members of a tuple, list, or string using square bracket “array” notation
- *Note that all are 0 based…*

```
>>> tu = (23, 'abc', 4.56, (2,3), 'def')
>>> tu[1]      # Second item in the tuple.
 'abc'

>>> li = ["abc", 34, 4.34, 23]
>>> li[1]       # Second item in the list.
 34

>>> st = "Hello World"
>>> st[1]    # Second character in string.
 'e'
```

Positive and negative indices

```
>>> t = (23, 'abc', 4.56, (2,3), 'def')
```

Positive index: count from the left, starting with 0

```
>>> t[1]
'abc'
```

Negative index: count from right, starting with –1

```
>>> t[-3]
4.56
```

Slicing: return copy of a subset

```
>>> t = (23, 'abc', 4.56,
(2,3), 'def')
```

Return a copy of the container with a subset of the original members. Start copying at the first index, and stop copying *before* second.

```
>>> t[1:4]
('abc', 4.56, (2,3))
```

Negative indices count from end

```
>>> t[1:-1]
('abc', 4.56, (2,3))
```

Slicing: return copy of a =subset

```
>>> t = (23, 'abc', 4.56,
(2,3), 'def')
```

Omit first index to make copy starting from beginning of the container

```
>>> t[:2]
(23, 'abc')
```

Omit second index to make copy starting at first index and going to end

```
>>> t[2:]
(4.56, (2,3), 'def')
```

Copying the Whole Sequence

- [:] makes a *copy* of an entire sequence

```
>>> t[:]
(23, 'abc', 4.56, (2,3), 'def')
```

- Note the difference between these two lines for mutable sequences

```
>>> l2 = l1 # Both refer to 1 ref,
            # changing one affects
  both
>>> l2 = l1[:] # Independent
  copies, two refs
```

The 'in' Operator

- Boolean test whether a value is inside a container:

```
>>> t = [1, 2, 4, 5]
>>> 3 in t
False
>>> 4 in t
True
>>> 4 not in t
False
```

- For strings, tests for substrings

```
>>> a = 'abcde'
>>> 'c' in a
True
>>> 'cd' in a
True
>>> 'ac' in a
False
```

- Be careful: the *in* keyword is also used in the syntax of *for loops* and *list comprehensions*

The + Operator

The + operator produces a *new* tuple, list, or string whose value is the concatenation of its arguments.

```
>>> (1, 2, 3) + (4, 5, 6)
 (1, 2, 3, 4, 5, 6)

>>> [1, 2, 3] + [4, 5, 6]
 [1, 2, 3, 4, 5, 6]

>>> "Hello" + " " + "World"
 'Hello World'
```

The * Operator

- The * operator produces a *new* tuple, list, or string that “repeats” the original content.

```
>>> (1, 2, 3) * 3
(1, 2, 3, 1, 2, 3, 1, 2, 3)

>>> [1, 2, 3] * 3
[1, 2, 3, 1, 2, 3, 1, 2, 3]

>>> “Hello” * 3
‘HelloHelloHello’
```

Functions

A function is a block of organized, reusable code that is used to perform a single, related action. Functions provide better modularity for your application and a high degree of code reusing.

simple rules to define a function in Python.

- ? Function blocks begin with the keyword def followed by the function name and parentheses (( )).

- ? Any input parameters or arguments should be placed within these parentheses. You can also define parameters inside these parentheses.

- ? The code block within every function starts with a colon (:) and is indented.

- ? The statement return [expression] exits a function, optionally passing back an expression to the caller. A return statement with no arguments is the same as return None.

»

```
def functionname( parameters ):
    function_suite
return [expression]
```

EXAMPLE

```
def printme( str ):
  print (str)
  return
```

```
# Function definition is here
def printme( str ):
   "This prints a passed string into this function"
   print (str)
   return
# Now you can call printme function
printme("This is first call to the user defined function!")

printme("Again second call to the same function")
```

Pass by Reference vs Value

```
# Function definition is here
def changeme( mylist ):
"This changes a passed list into this function"
print ("Values inside the function before change: ", mylist) mylist[2]=50
print ("Values inside the function after change: ", mylist)

   return
# Now you can call changeme function
mylist = [10,20,30]
changeme( mylist )
print ("Values outside the function: ", mylist)
```

Values inside the function before change: [10, 20, 30] Values inside the function after change: [10, 20, 50]
Values outside the function: [10, 20, 50]

»

```
 # Function definition is here
 def changeme( mylist ):
"This changes a passed list into this function"
mylist = [1,2,3,4] # This would assi new reference in mylist print ("Values inside the function: ", mylist)
return

# Now you can call changeme function
mylist = [10,20,30]
changeme( mylist )
print ("Values outside the function: ", mylist)
```

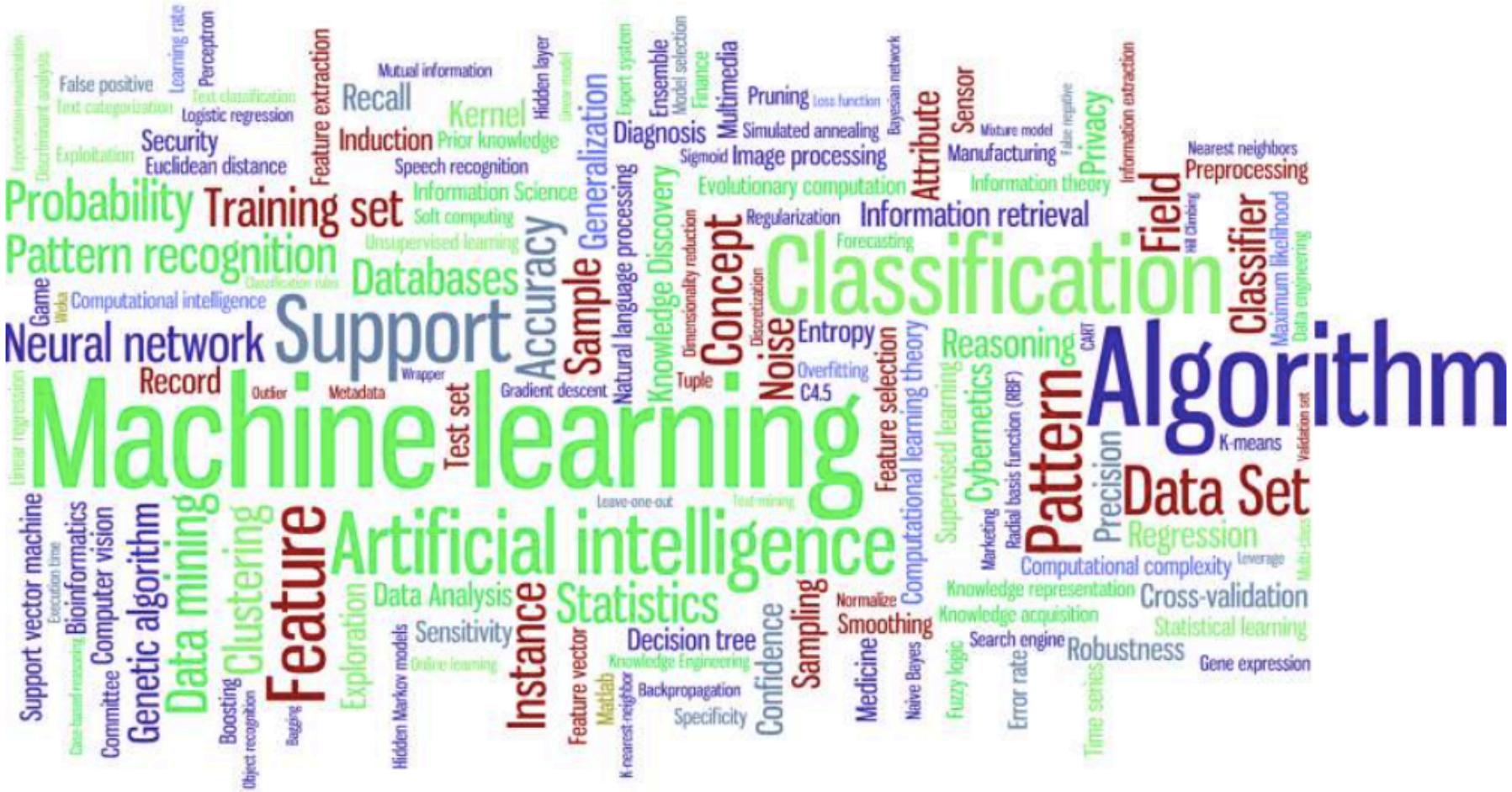
Values inside the function: [1, 2, 3, 4] Values outside the function: [10, 20, 30]

»

MODULES

- import numpy
- import pandas
- from sklearn.ensemble import RandomForestClassifier

Introduction to Machine Learning



Machine Learning

“Learning is any process by which a system improves performance from experience.”

“Machine Learning is concerned with computer programs that automatically improve their performance through experience. “

Why Machine Learning?

- Develop systems that can automatically adapt and customize themselves to individual users.
  - Personalized news or mail filter
- Discover new knowledge from large databases (data mining).
  - Market basket analysis (e.g. diapers and beer)

Why Machine Learning?

Ability to mimic human and replace certain monotonous tasks which require some intelligence.

Like recognizing handwritten characters.

Develop systems that are too difficult/ expensive to construct manually because they require specific detailed skills or knowledge tuned to a specific task (knowledge engineering bottleneck).

Why now?

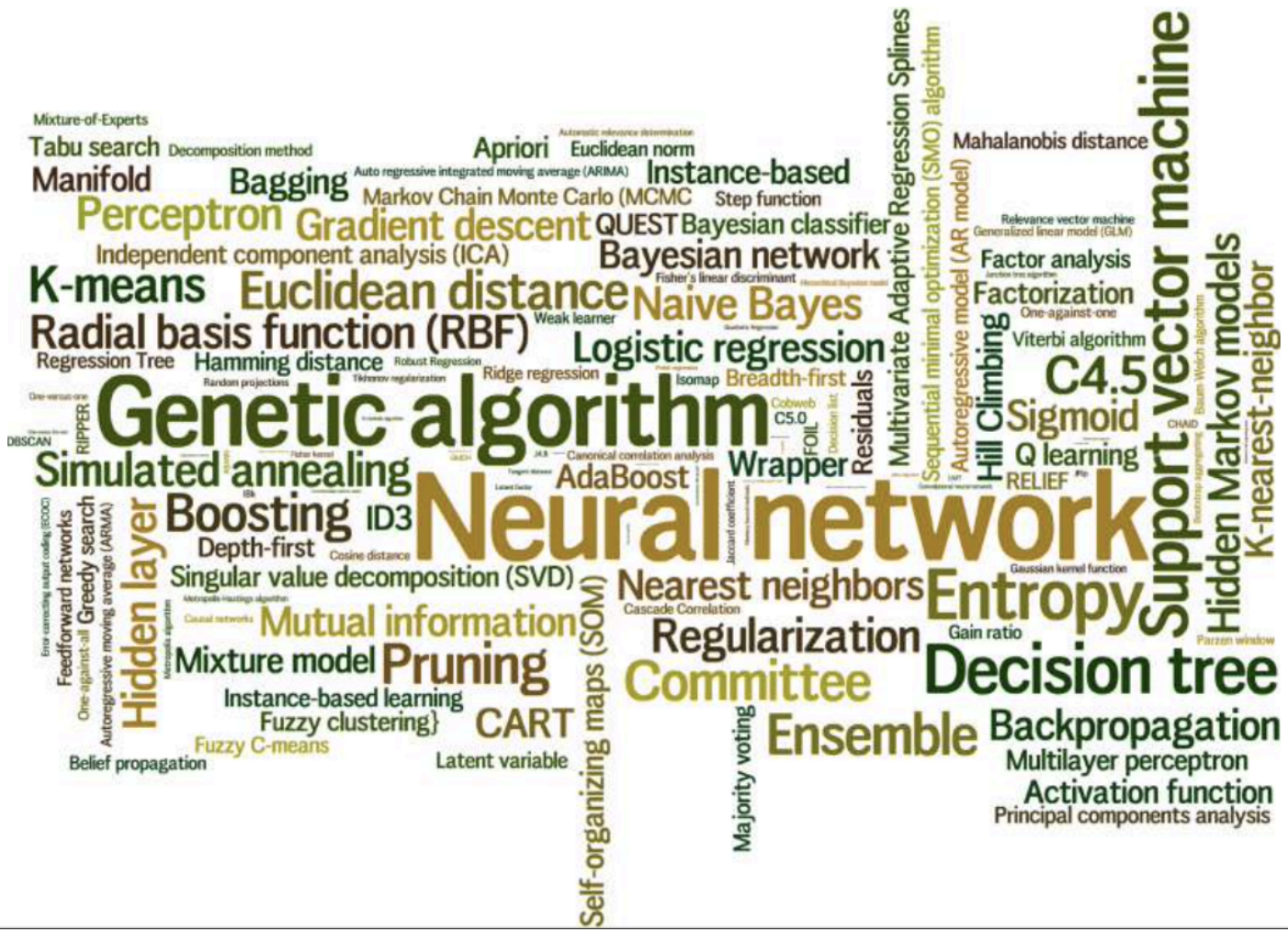
- Flood of available data (especially with the advent of the Internet)
- Increasing computational power
- Growing progress in available algorithms and

theory developed by researchers
- Increasing support from industries

ML Applications

Preprocessing

Learning Algorithms



Types of Machine Learning Problems

Supervised

Unsupervised

Reinforcement

Supervised Learning

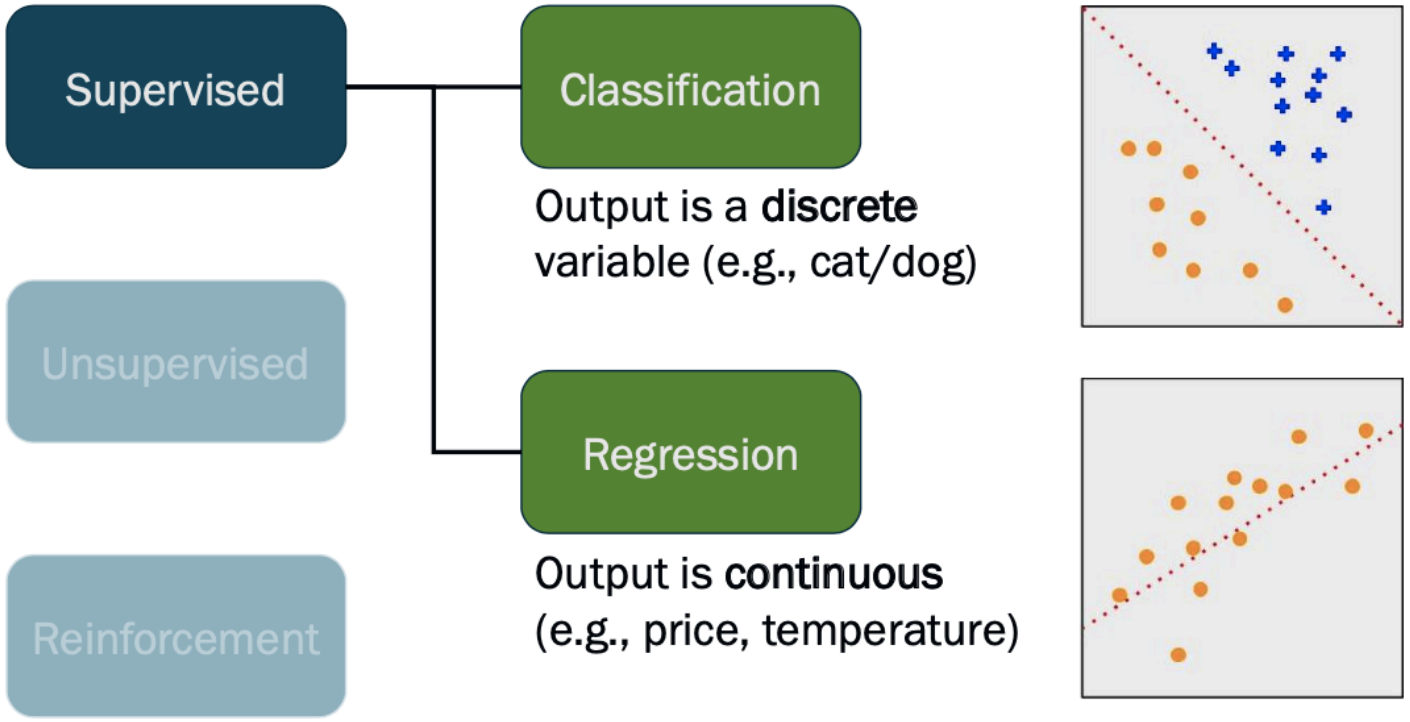
Learn through examples of which we know the desired output (what we want to predict).

Is this a cat or a dog?

Are these emails spam or not?

Predict the market value of houses, given the square meters, number of rooms, neighborhood, etc.

Supervised Learning



UnSupervised Learning

There is no desired output. Learn something about the data. Latent relationships.

I have photos and want to put them in 20 groups.

I want to find anomalies in the credit card usage patterns of my customers.

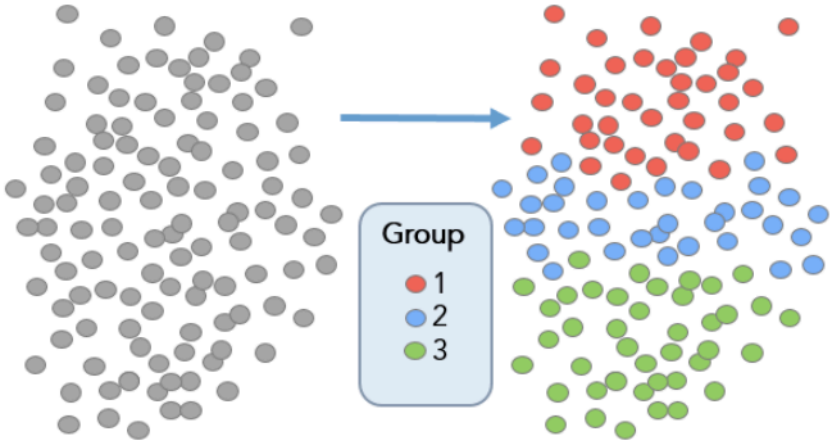
UnSupervised Learning

Supervised

Unsupervised

Reinforcement

Useful for learning structure in the data (**clustering**), hidden correlations, reduce dimensionality, etc.



Reinforcement Learning

An agent interacts with an environment and watches the result of the interaction.

Environment gives feedback via a positive or negative reward signal.

Steps to Solve a Machine Learning Problem

Data Gathering

Might depend on human work
• Manual labeling for supervised learning. • Domain knowledge. Maybe even experts.

May come for free, or “sort of” • E.g., Machine Translation.

The more the better: Some algorithms need large amounts of data to be useful (e.g., neural networks).

The quantity and quality of data dictate the model accuracy.

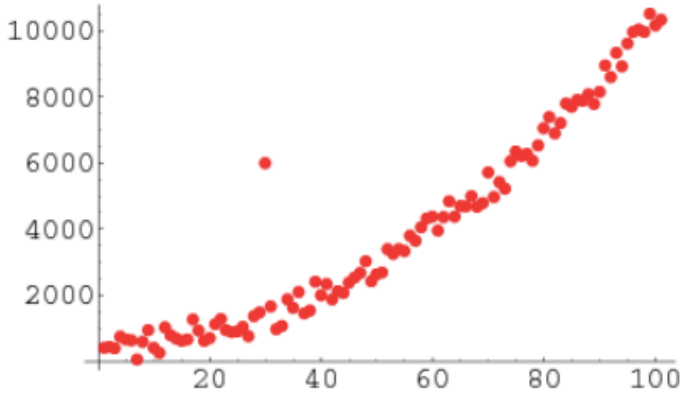
Data Preprocessing

Is there anything wrong with the data?

Missing values
Outliers
Bad encoding (for text)

Wrongly-labeled examples

Biased data
Do I have many more samples of one class than the rest?
Need to fix/remove data?



Feature Engineering

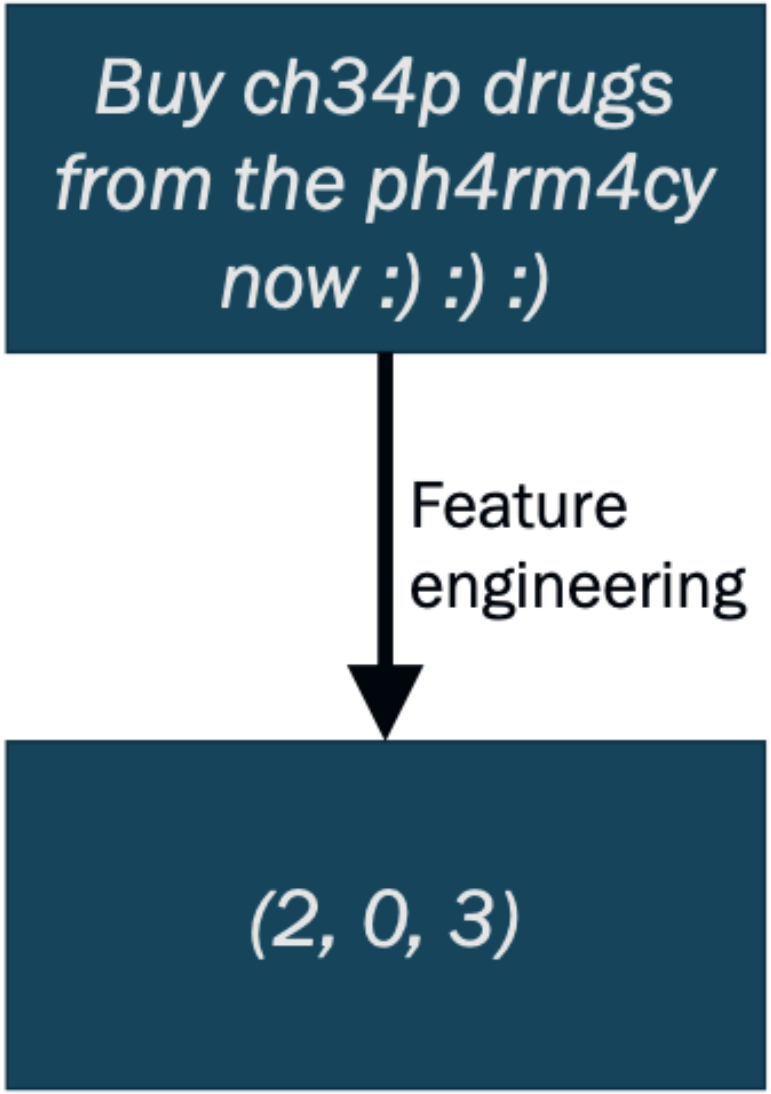
What is a feature?

A feature is an individual measurable property of a phenomenon being observed

Our inputs are represented by a set of features. To classify spam email, features could be:

- Number of words that have been ch4ng3d like this.
- Language of the email (0=English, 1=Spanish)
- Number of emojis

Feature Engineering



Feature Engineering

Extract more information from existing data, not adding “new” data per-se • Making it more useful
• With good features, most algorithms can learn faster

It can be an art
• Requires thought and knowledge of the data

Two steps:

Variable transformation (e.g., dates into weekdays, normalizing)

Feature creation (e.g., n-grams for texts, if word is capitalized to detect names, etc.)

Algorithm Selection & Training

Supervised

- Linear classifier
- Linear Regression
- Naive Bayes
- Support Vector Machines (SVM)
- Decision Tree
- Random Forests
- k-Nearest Neighbors
- Neural Networks (Deep learning)

Unsupervised

- PCA
- t-SNE
- k-means • DBSCAN

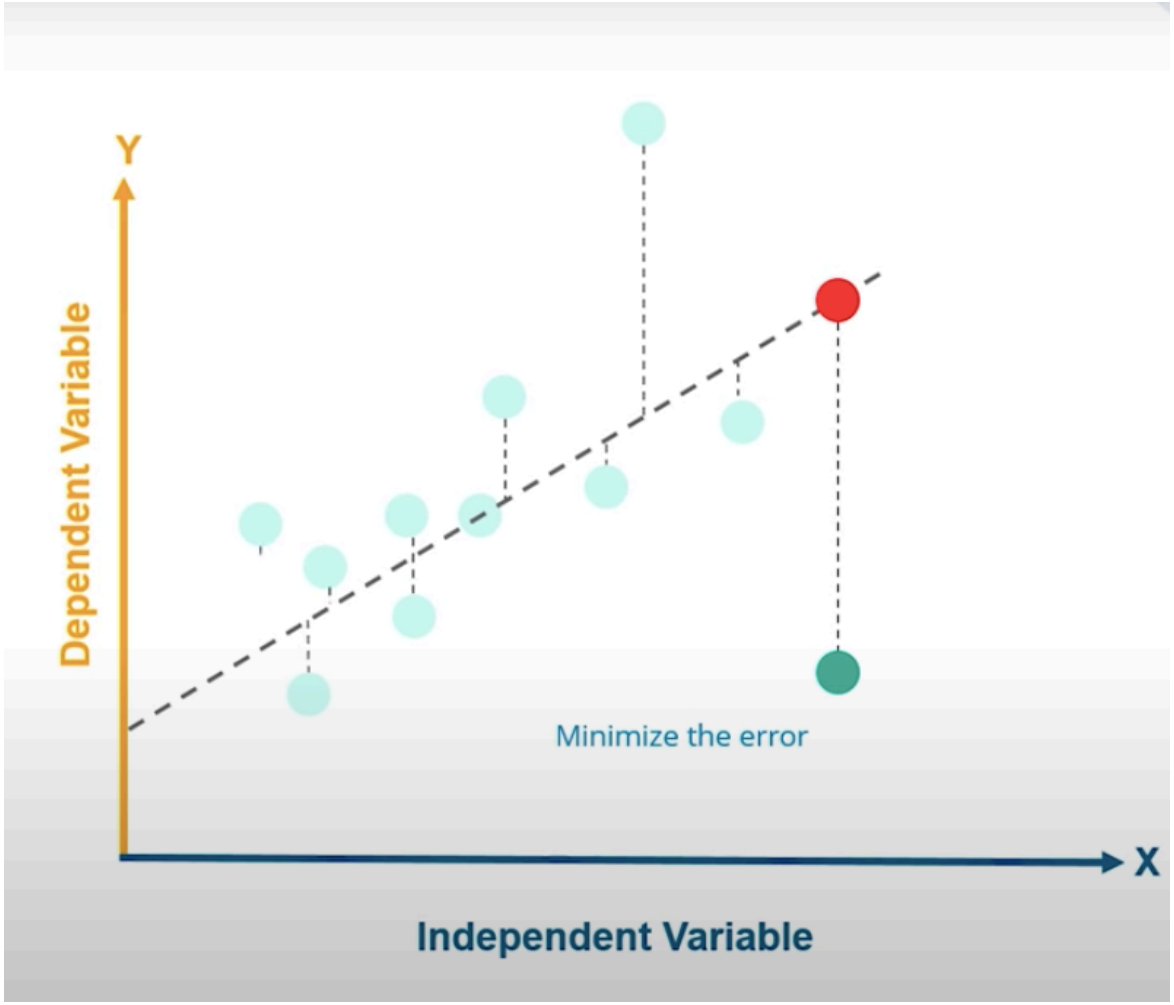
Reinforcement

- SARSA–λ
- Q-Learning

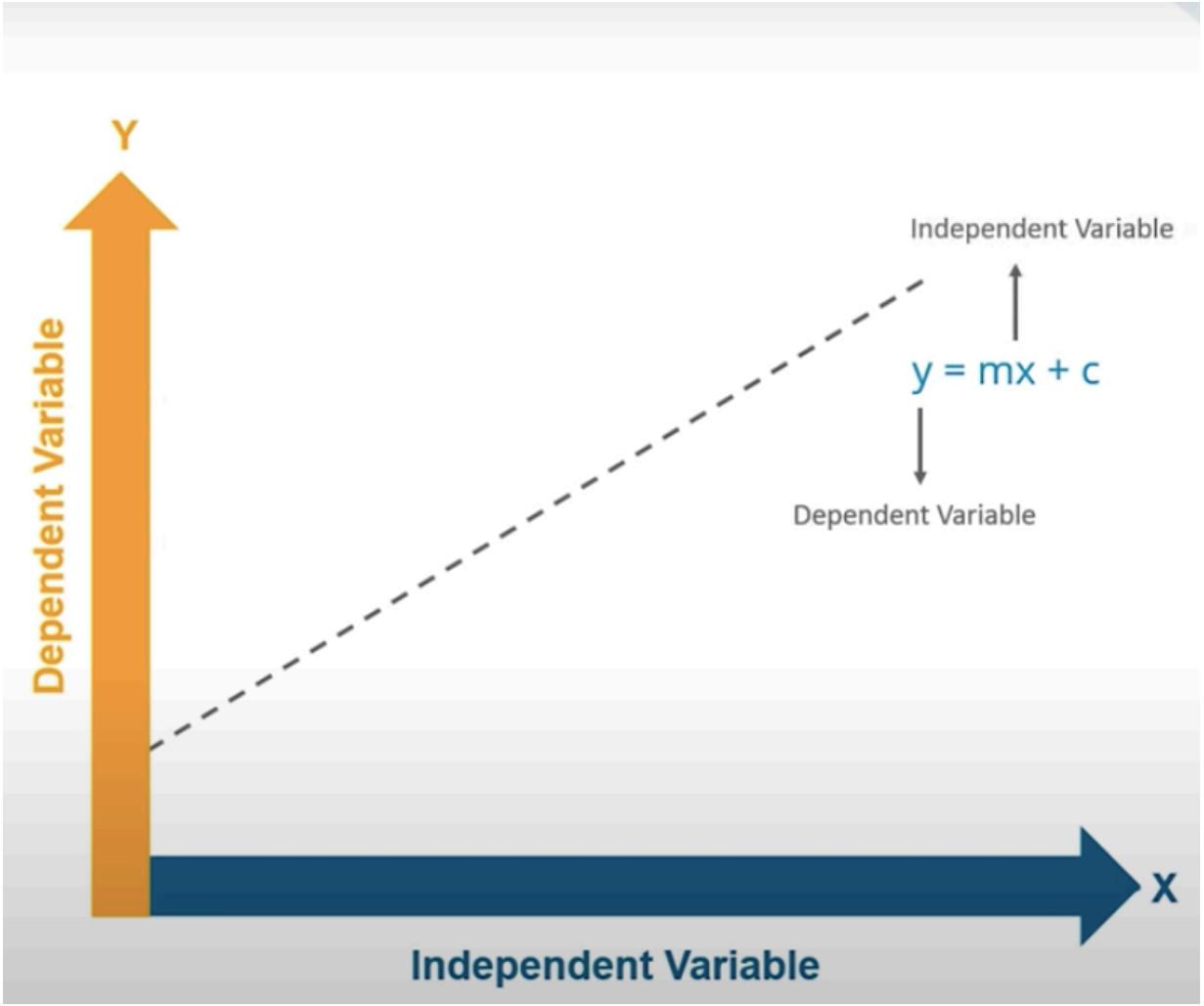
Reinforcement Learning

- Positive
- Negative

Linear Regression

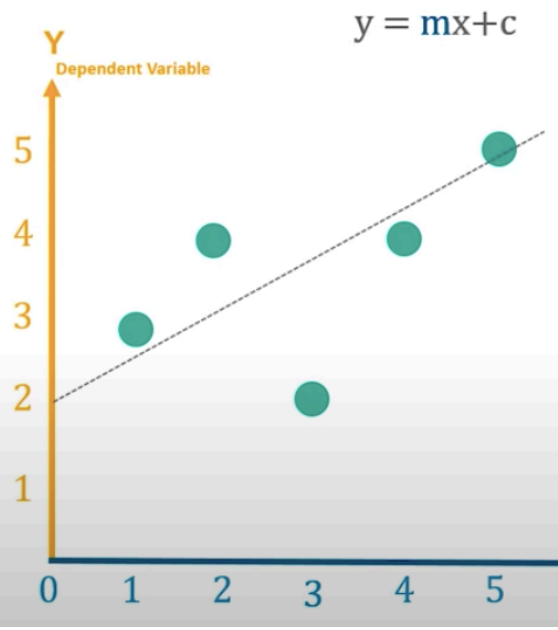


Linear Regression



Linear Regression

Understanding Linear Regression Algorithm



x	y	$x-\bar{x}$	$y-\bar{y}$	$(x-\bar{x})^2$	$(x-\bar{x})(y-\bar{y})$
1	3	-2	-0.6	4	1.2
2	4	-1	0.4	1	-0.4
3	2	0	-1.6	0	0
4	4	1	0.4	1	0.4
5	5	2	1.4	4	2.8
mean	3	3.6		$\Sigma = 10$	$\Sigma = 4$

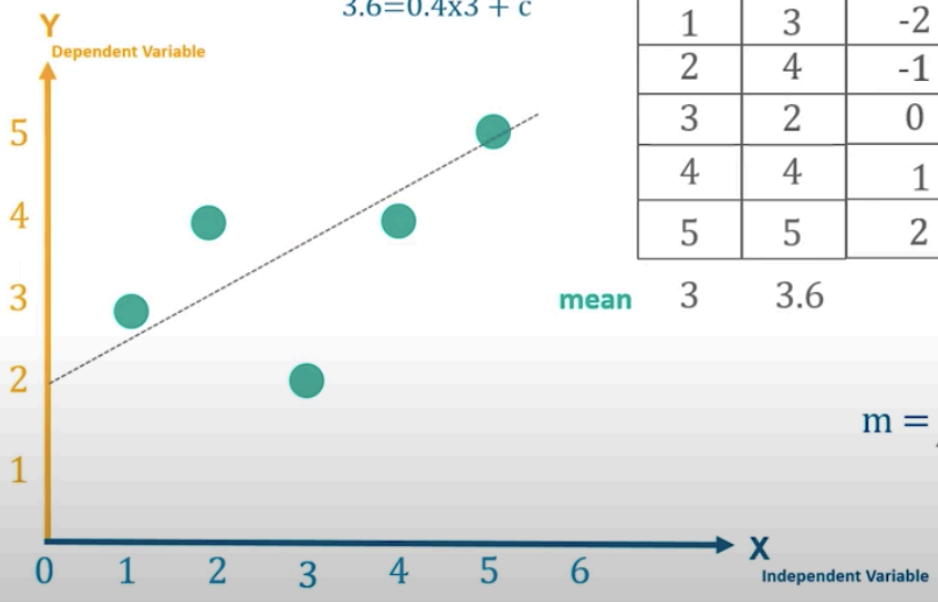
$$m = \frac{\sum (x-\bar{x})(y-\bar{y})}{\sum (x-\bar{x})^2}$$

Linear Regression

Understanding Linear Regression Algorithm

$y = mx+c$

$3.6=0.4x3 + c$



x	y	$x-\bar{x}$	$y-\bar{y}$	$(x-\bar{x})^2$	$(x-\bar{x})(y-\bar{y})$
1	3	-2	-0.6	4	1.2
2	4	-1	0.4	1	-0.4
3	2	0	-1.6	0	0
4	4	1	0.4	1	0.4
5	5	2	1.4	4	2.8
mean	3	3.6		$\Sigma = 10$	$\Sigma = 4$

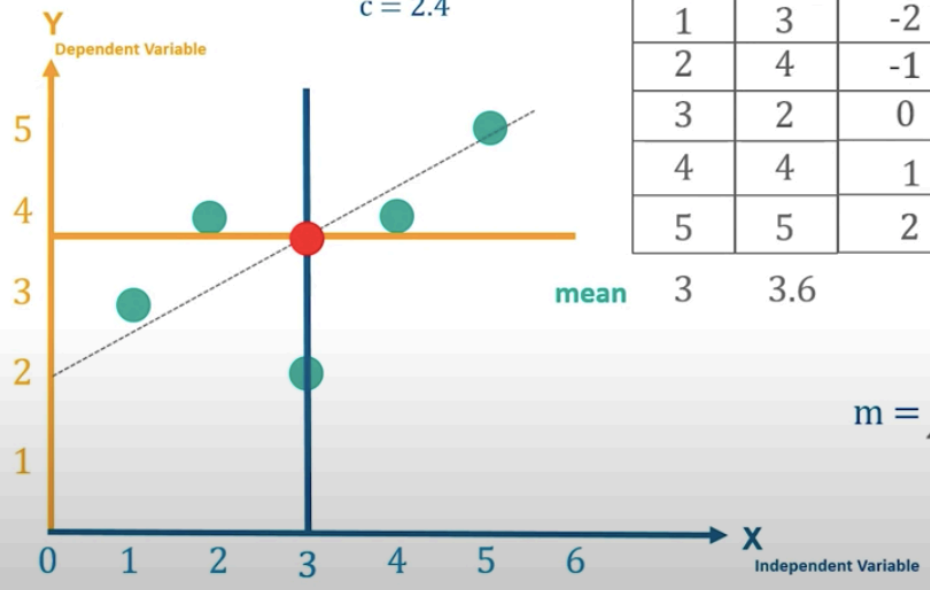
$$m = \sum \frac{(x-\bar{x})(y-\bar{y})}{(x-\bar{x})^2}$$

Linear Regression

Understanding Linear Regression Algorithm

$y = mx+c$

$c = 2.4$



x	y	$x-\dot{x}$	$y-\dot{y}$	$(x-\dot{x})^2$	$(x-\dot{x})(y-\dot{y})$
1	3	-2	-0.6	4	1.2
2	4	-1	0.4	1	-0.4
3	2	0	-1.6	0	0
4	4	1	0.4	1	0.4
5	5	2	1.4	4	2.8
mean	3	3.6		$\Sigma = 10$	$\Sigma = 4$

$$m = \sum \frac{(x-\dot{x})(y-\dot{y})}{(x-\dot{x})^2} = \frac{4}{10}$$

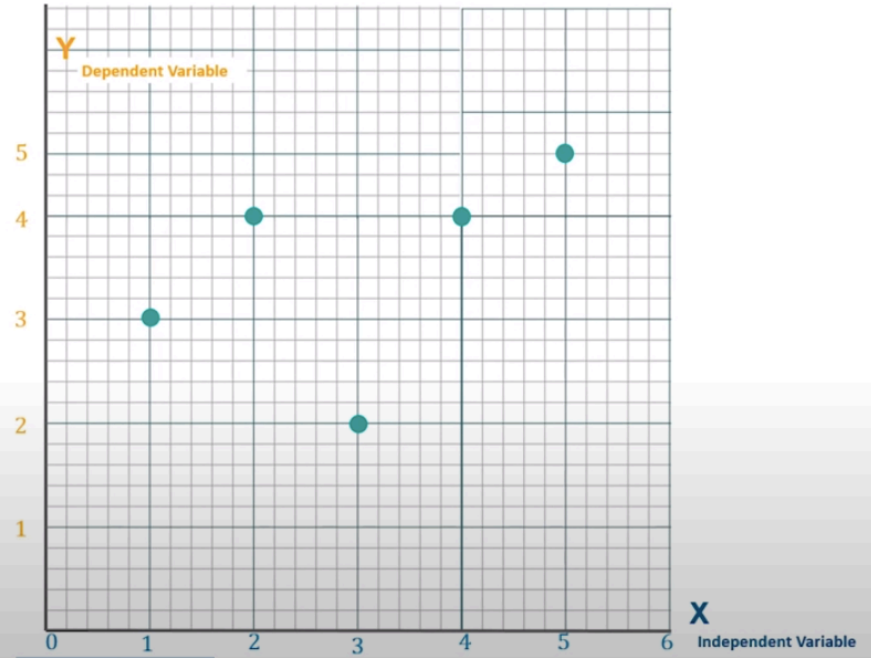
$m = 0.4$

$c = 2.4$

$y = 0.4x + 2.4$

Linear Regression

Mean Square Error



$m = 0.4$
$c = 2.4$
$y = 0.4x + 2.4$

For given $m = 0.4$ & $c = 2.4$, lets predict values for y for $x = \{1,2,3,4,5\}$

$y = 0.4 \times 1 + 2.4 = 2.8$
$y = 0.4 \times 2 + 2.4 = 3.2$
$y = 0.4 \times 3 + 2.4 = 3.6$
$y = 0.4 \times 4 + 2.4 = 4.0$
$y = 0.4 \times 5 + 2.4 = 4.4$

Linear Regression

R-Squared Value - Statistical measure of how close the data are to the fitted regression line.

Numpy

- Setup - pip install numpy
- To import - import numpy

Numpy

- NumPy is a Python package.
- It stands for 'Numerical Python'.
- It is a library consisting of multidimensional array objects and a collection of routines for processing of array.
- Replacement for MatLab.

Numpy

Using NumPy, a developer can perform the following operations −

- Mathematical and logical operations on arrays.
- Fourier transforms and routines for shape manipulation.
- Operations related to linear algebra. NumPy has in-built functions for linear algebra and random number generation.

Numpy

```
#one dimension
import numpy as np
arr = np.array([1,2,3])
print (arr)

# more than one dimensions
arr = np.array([[1, 2], [3, 4]])
print (arr)
```

Numpy

```
# minimum dimensions
import numpy as np
arr = np.array([1, 2, 3,4,5], ndmin = 2)
print (arr)

»
```

Numpy

```
#shape
arr = np.array([[1,2,3],[4,5,6]])
print (arr.shape)

#reshape
arr = np.array([[1,2,3],[4,5,6]])
b = arr.reshape(3,2)
print b
```

»

Numpy

```
#print 0 - 23
a = np.arange(24)
a.ndim

# dtype of array is int8 (1 byte)
x = np.array([1,2,3,4,5], dtype = np.int8)
print x.itemsize

# dtype of array is now float32 (4 bytes)
x = np.array([1,2,3,4,5], dtype = np.float32)
print x.itemsize
```

Numpy

- # print 0 to 49 , with a gap of 2
- #e.g. - 0,2,4,6,…48
- arr = np.arange(0,50,2)

- # print 0 to 49 , with a gap of 4
- #e.g. - 0,4,8,…48
- arr = np.arange(0,50,4)

-

Numpy

```
# To iterate
for i in np.nditer(arr):
    print(i)

# Transpose of Matrix
if Matrix is mat
then mat.T
```

Numpy

```
a = np.arange(0,60,5)
a = a.reshape(3,4)

#print 'Original array is:'
print (a)

#print 'Transpose of the original array is:'
b = a.T
print (b)

#print 'Modified array is:'
for x in np.nditer(b):
    print(x),
```

Numpy

```
arr = np.array([[1,2],[3,4]])

#print 'First array:'
print (arr)

b = np.array([[5,6],[7,8]])
#print 'Second array:'
print (b)

# both the arrays are of same dimensions
#print ‘Joining the two arrays along axis 0:'
print (np.concatenate((a,b)) )

#print 'Joining the two arrays along axis 1:'
print (np.concatenate((a,b),axis = 1))
```

»

Numpy

```
arr = np.array([0,30,45,60,90])

#print 'Sine of different angles:'
# Convert to radians by multiplying with pi/180
print (np.sin(a*np.pi/180))

#print 'Cosine values for angles in array:'
print (np.cos(a*np.pi/180))

#print 'Tangent values for given angles:'
print (np.tan(a*np.pi/180))
```

- # Inverse
- # arcsin , arccos, arctan

Numpy

```
arr = np.arange(9, dtype = np.float_).reshape(3,3)

#print 'First array:'
print (a)

#print 'Second array:'
b = np.array([10,10,10])
print (b)

#print 'Add the two arrays:'
print np.add(a,b)

#print 'Subtract the two arrays:'
print np.subtract(a,b)

#print 'Multiply the two arrays:'
print np.multiply(a,b)

#print 'Divide the two arrays:'
print np.divide(a,b)
»
```

Numpy

```
arr = np.array([[30,65,70],[80,95,10],[50,90,60]])

#print 'Our array is:'
print (arr)

#print 'Applying median() function:'
print np.median(arr)

#print 'Applying median() function along axis 0:'
print np.median(arr, axis = 0)

#print 'Applying median() function along axis 1:'
print np.median(arr, axis = 1)

# same syntax for mean, just replace median with mean
```

Numpy

```
std = sqrt(mean(abs(x - x.mean())**2))
#print np.std([1,2,3,4])
#print np.var([1,2,3,4])


»#MATRIX
import numpy as np
import numpy.matlib
#print np.matlib.empty((2,2)) # filled with random data

#print np.matlib.zeros((2,2)) # filled with zeros
#print np.matlib.ones((2,2)) # filled with ones
#print np.matlib.identity(5, dtype = float)
```

Numpy

» Linear Regression

Function & Description

1. dot

Dot product of the two arrays

2. vdot

Dot product of the two vectors

3. inner

Inner product of the two arrays

Numpy

» Linear Regression

Function & Description

4. matmul
Matrix product of the two arrays

5. determinant
Computes the determinant of the array

6. solve
Solves the linear matrix equation

7. inv
Finds the multiplicative inverse of the matrix

»

Numpy

```
#To calculate Determinant
#print 6*(-2*7 - 5*8) - 1*(4*7 - 5*2) + 1*(4*8 - -2*2)

import numpy as np
b = np.array([[6,1,1], [4, -2, 5], [2,8,7]])
#print b
#print np.linalg.det(b)
```

»

Pandas

- Pandas is an open-source, BSD-licensed Python library.
- It provides high-performance, easy-to-use data structures and data analysis tools for the Python programming language.
- Python with Pandas is used in a wide range of fields including academic and commercial domains including finance, economics, Statistics, analytics, etc.
- The name Pandas is derived from the word Panel Data – an Econometrics from Multidimensional data.

Pandas

Key Features of Panda

- Fast and efficient DataFrame object with default and customized indexing.
- Tools for loading data into in-memory data objects from different file formats.
- Data alignment and integrated handling of missing data.
- Reshaping and pivoting of date sets.
- Label-based slicing, indexing and subsetting of large data sets.
- Columns from a data structure can be deleted or inserted.
- Group by data for aggregation and transformations.
- High performance merging and joining of data.
- Time Series functionality.

Pandas

Setup - pip install pandas

» **Import** - import pandas

Pandas

Pandas deals with the following three data structures –

- Series
- DataFrame
- Panel

These data structures are built on top of Numpy array, which means they are fast.

»

Pandas

The best way to think of these data structures is that the higher dimensional data structure is a container of its lower dimensional data structure. For example, DataFrame is a container of Series, Panel is a container of DataFrame.

All Pandas data structures are value mutable (can be changed) and except Series all are size mutable. Series is size immutable.

Pandas

Data Structure	Dimensions	Description
Series	1	1D labeled homogeneous array, sizeimmutable.
Data Frames	2	General 2D labeled, size-mutable tabular structure with potentially heterogeneously typed columns.
Panel	3	General 3D labeled, size-mutable array.

Pandas

Series

Series is a one-dimensional array like structure with homogeneous data. For example, the following series is a collection of integers 10, 23, 56, …

Key Points

- Homogeneous data
- Size Immutable
- Values of Data Mutable

Pandas

DataFrame

DataFrame is a two-dimensional array with heterogeneous data.

Key Points

- Heterogeneous data
- Size Mutable
- Data Mutable

»

Pandas

- DataFrame Example,

-

Name	Age	Gender	Rating
Steve	32	Male	3.45
Lia	28	Female	4.6
Vin	45	Male	3.9
Katie	38	Female	2.78

Pandas

Panel

Panel is a three-dimensional data structure with heterogeneous data. It is hard to represent the panel in graphical representation. But a panel can be illustrated as a container of DataFrame.

Key Points

- Heterogeneous data
- Size Mutable
- Data Mutable

Pandas

pandas.Series

A pandas Series can be created using the following constructor –

pandas.Series(data, index, dtype, copy)

```
#import the pandas library and aliasing as pd
import pandas as pd
s = pd.Series()
#print s
```

»

Pandas

Create a Series from ndarray

```
import numpy as np
data = np.array(['a','b','c','d'])
s = pd.Series(data)
#print s

data = np.array(['a','b','c','d'])
s = pd.Series(data,index=[100,101,102,103])
#print s
```

»

Pandas

Create a Series from dict

```
data = {'a' : 0., 'b' : 1., 'c' : 2.}
s = pd.Series(data)
print s
```

```
data = {'a' : 0., 'b' : 1., 'c' : 2.}
s = pd.Series(data,index=['b','c','d','a'])
print s
```

»

Pandas

Accessing Data from Series with Position

```
s = pd.Series([1,2,3,4,5],index = ['a','b','c','d','e'])

#retrieve the first element
print s[0]

s = pd.Series([1,2,3,4,5],index = ['a','b','c','d','e'])

#retrieve the first three element
print s[:3]

s = pd.Series([1,2,3,4,5],index = ['a','b','c','d','e'])

#retrieve the last three element
print s[-3:]
```

»

Pandas

```
s = pd.Series([1,2,3,4,5],index = ['a','b','c','d','e'])

#retrieve a single element using label
print s['a']

s = pd.Series([1,2,3,4,5],index = ['a','b','c','d','e'])

#retrieve multiple elements using label
print s[['a','c','d']]
```

»

Pandas

» DataFrame

A Data frame is a two-dimensional data structure, i.e., data is aligned in a tabular fashion in rows and columns.

Features of DataFrame

- Potentially columns are of different types
- Size – Mutable
- Labeled axes (rows and columns)
- Can Perform Arithmetic operations on rows and columns

Pandas

Create DataFrame

A pandas DataFrame can be created using various inputs like –

- Lists
- dict
- Series
- Numpy ndarrays
- Another DataFrame

Pandas

```
import pandas as pd
df = pd.DataFrame()  # Creates Empty Dataframe
print df

data = [1,2,3,4,5]
df = pd.DataFrame(data) #Creates DataFrame from List
print df

data = [['Rahul',10],['Raj',12],['Max',13]]
df = pd.DataFrame(data,columns=['Name','Age'])
print df
```

»

Pandas

```
d = {'one' : pd.Series([1, 2, 3], index=['a', 'b', 'c']),
   'two' : pd.Series([1, 2, 3, 4], index=['a', 'b', 'c', 'd'])}

df = pd.DataFrame(d) #df from dict of series
print df

d = {'one' : pd.Series([1, 2, 3], index=['a', 'b', 'c']),
   'two' : pd.Series([1, 2, 3, 4], index=['a', 'b', 'c', 'd'])}

df = pd.DataFrame(d)
print df [‘one'] # print selected columns
```

»

Pandas

```
d = {'one' : pd.Series([1, 2, 3], index=['a', 'b', 'c']),
   'two' : pd.Series([1, 2, 3, 4], index=['a', 'b', 'c', 'd'])}

df = pd.DataFrame(d)

# Adding a new column to an existing DataFrame object with column
label by passing new series

print ("Adding a new column by passing as Series:")
df['three']=pd.Series([10,20,30],index=['a','b','c'])
print df

print ("Adding a new column using the existing columns in DataFrame:")
df['four']=df['one']+df['three']

print df
```

»

Pandas

```
d = {'one' : pd.Series([1, 2, 3], index=['a', 'b', 'c']),
   'two' : pd.Series([1, 2, 3, 4], index=['a', 'b', 'c', 'd']),
   'three' : pd.Series([10,20,30], index=['a','b','c'])}

df = pd.DataFrame(d)
print ("Our dataframe is:")
print df

# using del function
print ("Deleting the first column using DEL function:")
del df['one']
print df

# using pop function
print ("Deleting another column using POP function:")
df.pop('two')
print df
```

Pandas

Selection by Label
Rows can be selected by passing row label to a loc function.

```
d = {'one' : pd.Series([1, 2, 3], index=['a', 'b', 'c']),
   'two' : pd.Series([1, 2, 3, 4], index=['a', 'b', 'c', 'd'])}

df = pd.DataFrame(d)
print df.loc['b']
```

Selection by integer location
Rows can be selected by passing integer location to an iloc function.

```
d = {'one' : pd.Series([1, 2, 3], index=['a', 'b', 'c']),
   'two' : pd.Series([1, 2, 3, 4], index=['a', 'b', 'c', 'd'])}

df = pd.DataFrame(d)
print df.iloc[2]
```

»

Pandas

Slice Rows
Multiple rows can be selected using ‘ : ’ operator.

```
d = {'one' : pd.Series([1, 2, 3], index=['a', 'b', 'c']),
   'two' : pd.Series([1, 2, 3, 4], index=['a', 'b', 'c', 'd'])}

df = pd.DataFrame(d)
print df[2:4]
```

Addition of Rows
Add new rows to a DataFrame using the append function. This function will append the rows at the end.

```
df = pd.DataFrame([[1, 2], [3, 4]], columns = ['a','b'])
df2 = pd.DataFrame([[5, 6], [7, 8]], columns = ['a','b'])

df = df.append(df2)
print df
```

»

Pandas

Deletion of Rows

Use index label to delete or drop rows from a DataFrame. If label is duplicated, then multiple rows will be dropped.

If you observe, in the above example, the labels are duplicate. Let us drop a label and will see how many rows will get dropped.

```
df = pd.DataFrame([[1, 2], [3, 4]], columns = ['a','b'])
df2 = pd.DataFrame([[5, 6], [7, 8]], columns = ['a','b'])

df = df.append(df2)

# Drop rows with label 0
df = df.drop(0)

print df
```

»

Pandas

Series Basic Functionality

»

Sr.No.	Attribute or Method & Description
1	**axes** Returns a list of the row axis labels
2	**dtype** Returns the dtype of the object.
3	**empty** Returns True if series is empty.
4	**ndim** Returns the number of dimensions of the underlying data, by definition 1.
5	**size** Returns the number of elements in the underlying data.
6	**values** Returns the Series as ndarray.
7	**head()** Returns the first n rows.
8	**tail()** Returns the last n rows.

Pandas

```
s = pd.Series(np.random.randn(4))
print s
```

Functions & Description

Let us now understand the functions under Descriptive Statistics in Python Pandas. The following table list down the important functions –

»

Pandas

Sr.No.	Function	Description
1	count()	Number of non-null observations
2	sum()	Sum of values
3	mean()	Mean of Values
4	median()	Median of Values
5	mode()	Mode of values
6	std()	Standard Deviation of the Values
7	min()	Minimum Value
8	max()	Maximum Value
9	abs()	Absolute Value
10	prod()	Product of Values
11	cumsum()	Cumulative Sum
12	cumprod()	Cumulative Product

Pandas

```
#Create a Dictionary of series
d =
{'Name':pd.Series(['Rahul','Raj','Aarav','Aditya','Satpal','Sourabh']),
   'Age':pd.Series([25,26,25,23,30,29]),
   'Rating':pd.Series([4.23,3.24,3.98,2.56,3.20,4.6])
}

#Create a DataFrame
df = pd.DataFrame(d)
print (df)

» print(df.sum())

»
```

Pandas

Summarizing Data
The describe() function computes a summary of statistics pertaining to the DataFrame columns.

print df.describe()

- object – Summarizes String columns
- number – Summarizes Numeric columns
- all – Summarizes all columns together (Should not pass it as a list value)
-

Pandas

Iterating a DataFrame
Iterating a DataFrame gives column names

To iterate over the rows of the DataFrame, we can use the following functions –

- iteritems() – to iterate over the (key,value) pairs
- iterrows() – iterate over the rows as (index,series) pairs
- itertuples() – iterate over the rows as namedtuples
-

Pandas

#iteritems()

```
df = pd.DataFrame(np.random.randn(4,3),columns=['col1','col2','col3'])
for key,value in df.iteritems():
   print key,value
```

iterrows()

```
df = pd.DataFrame(np.random.randn(4,3),columns = ['col1','col2','col3'])
for row_index,row in df.iterrows():
   print row_index,row
```

itertuples()

```
df = pd.DataFrame(np.random.randn(4,3),columns = ['col1','col2','col3'])
for row in df.itertuples():
    print row
```

»

Pandas

Sorting Algorithm

sort_values() provides a provision to choose the algorithm from mergesort, heapsort and quicksort. Mergesort is the only stable algorithm.

```
unsorted_df = pd.DataFrame({'col1':[2,1,1,1],'col2':
[1,3,2,4]})
sorted_df =
unsorted_df.sort_values(by='col1' ,kind='mergesort')

print sorted_df
```

»

Pandas

Working with Text Data

»

Sr.No	Function & Description
1	**lower()** Converts strings in the Series/Index to lower case.
2	**upper()** Converts strings in the Series/Index to upper case.
3	**len()** Computes String length().
4	**strip()** Helps strip whitespace(including newline) from each string in the Series/index from both the sides.
5	**split(' ')** Splits each string with the given pattern.
6	**cat(sep=' ')** Concatenates the series/index elements with given separator.

Pandas

Working with Text Data

»

7	**get_dummies()** Returns the DataFrame with One-Hot Encoded values.
8	**contains(pattern)** Returns a Boolean value True for each element if the substring contains in the element, else False.
9	**replace(a,b)** Replaces the value **a** with the value **b**.
10	**repeat(value)** Repeats each element with specified number of times.
11	**count(pattern)** Returns count of appearance of pattern in each element.
12	**startswith(pattern)** Returns true if the element in the Series/Index starts with the pattern.

Pandas

Working with Text Data

»

13	**endswith(pattern)** Returns true if the element in the Series/Index ends with the pattern.
14	**find(pattern)** Returns the first position of the first occurrence of the pattern.
15	**findall(pattern)** Returns a list of all occurrence of the pattern.
16	**swapcase** Swaps the case lower/upper.
17	**islower()** Checks whether all characters in each string in the Series/Index in lower case or not. Returns Boolean
18	**isupper()** Checks whether all characters in each string in the Series/Index in upper case or not. Returns Boolean.
19	**isnumeric()** Checks whether all characters in each string in the Series/Index are numeric. Returns Boolean.

Pandas

```
s = pd.Series(['Dog', 'Cat', 'Elephant', 'Lion','Tiger'])

print s.str.lower()

print s.str.strip()

print s.str.len()

print s.str.split(' ')

print s.str.cat(sep='_')

print s.str.repeat(2)

print s.str.count('m')
```

Pandas

Sr.No	Indexing & Description
1	**.loc()** Label based
2	**.iloc()** Integer based
3	**.ix()** Both Label and Integer based

Pandas

.loc()

Pandas provide various methods to have purely **label based indexing**. When slicing, the start bound is also included. Integers are valid labels, but they refer to the label and not the position.

.loc() has multiple access methods like –

- A single scalar label
- A list of labels
- A slice object
- A Boolean array

loc takes two single/list/range operator separated by ','. The first one indicates the row and the second one indicates columns.

```
df = pd.DataFrame(np.random.randn(8, 4),
index = ['a','b','c','d','e','f','g','h'], columns = ['A', 'B', 'C', 'D'])

#select all rows for a specific column
print df.loc[:,'A']
```

»

Pandas

.iloc()

Pandas provide various methods in order to get purely integer based indexing. Like python and numpy, these are 0-based indexing.

The various access methods are as follows –

- An Integer
- A list of integers
- A range of values

```
df = pd.DataFrame(np.random.randn(8, 4), columns = ['A', 'B', 'C', 'D'])

# select all rows for a specific column
print df.iloc[:4]
```

»

Pandas

.ix()

Besides pure label based and integer based, Pandas provides a hybrid method for selections and subsetting the object using the .ix() operator.

```
df = pd.DataFrame(np.random.randn(8, 4), columns =
['A', 'B', 'C', 'D'])

# Integer slicing
print df.ix[:4]
```

»

Pandas

Missing Data

Let us consider an online survey for a product. Many a times, people do not share all the information related to them. Few people share their experience, but not how long they are using the product; few people share how long they are using the product, their experience but not their contact information. Thus, in some or the other way a part of data is always missing, and this is very common in real time.

```
df = pd.DataFrame(np.random.randn(5, 3), index=['a', 'c', 'e', 'f',
'h'],columns=['one', 'two', 'three'])

df = df.reindex(['a', 'b', 'c', 'd', 'e', 'f', 'g', 'h'])

print df
```

Pandas

```
df = pd.DataFrame(np.random.randn(5, 3), index=['a', 'c', 'e', 'f',
'h'],columns=['one', 'two', 'three'])

df = df.reindex(['a', 'b', 'c', 'd', 'e', 'f', 'g', 'h'])

print df['one'].isnull()


df = pd.DataFrame(np.random.randn(5, 3), index=['a', 'c', 'e', 'f',
'h'],columns=['one', 'two', 'three'])

df = df.reindex(['a', 'b', 'c', 'd', 'e', 'f', 'g', 'h'])

print df['one'].notnull()
»
```

Pandas

Calculations with Missing Data

- When summing data, NA will be treated as Zero
- If the data are all NA, then the result will be NA

```
df = pd.DataFrame(np.random.randn(5, 3), index=['a', 'c', 'e', 'f',
'h'],columns=['one', 'two', 'three'])

df = df.reindex(['a', 'b', 'c', 'd', 'e', 'f', 'g', 'h'])

print df['one'].sum()
```

»

Pandas

Cleaning / Filling Missing Data

Pandas provides various methods for cleaning the missing values. The fillna function can “fill in” NA values with non-null data in a couple of ways, which we have illustrated in the following sections.

Replace NaN with a Scalar Value

The following program shows how you can replace "NaN" with "0".

```
df = pd.DataFrame(np.random.randn(3, 3), index=['a', 'c',
'e'],columns=['one',
'two', 'three'])

df = df.reindex(['a', 'b', 'c'])

print df
print ("NaN replaced with '0':")
print df.fillna(0)
```

»

Pandas

Drop Missing Values

If you want to simply exclude the missing values, then use the dropna function along with the axis argument. By default, axis=0, i.e., along row, which means that if any value within a row is NA then the whole row is excluded.

```
df = pd.DataFrame(np.random.randn(5, 3), index=['a', 'c', 'e', 'f', 'h'],columns=['one', 'two', 'three'])

df = df.reindex(['a', 'b', 'c', 'd', 'e', 'f', 'g', 'h'])
print df.dropna()
```

»

Pandas

Replace Missing (or) Generic Values

Many times, we have to replace a generic value with some specific value. We can achieve this by applying the replace method.

Replacing NA with a scalar value is equivalent behavior of the fillna() function.

```
df = pd.DataFrame({'one':[10,20,30,40,50,2000], 'two':
[1000,0,30,40,50,60]})

print df.replace({1000:10,2000:60})
```

```
»
```