Network Programming Paradigm

Introduction

The Network paradigm involves thinking of computing in terms of a client, who is essentially in need of some type of information, and a server, who has lots of information and is just waiting to hand it out. Typically, a client will connect to a server and query for certain information. The server will go off and find the information and then return it to the client.

In the context of the Internet, clients are typically run on desktop or laptop computers attached to the Internet looking for information, whereas servers are typically run on larger computers with certain types of information available for the clients to retrieve. The Web itself is made up of a bunch of computers that act as Web servers; they have vast amounts of HTML pages and related data available for people to retrieve and browse. Web clients are used by those of us who connect to the Web servers and browse through the Web pages.

Network programming uses a particular type of network communication known as sockets. A socket is a software abstraction for an input or output medium of communication.

What is Socket?

- A socket is a software abstraction for an input or output medium of communication.
- Sockets allow communication between processes that lie on the same machine, or on different machines working in diverse environment and even across different continents.
- A socket is the most vital and fundamental entity. Sockets are the end-point of a two-way communication link.
- An endpoint is a combination of IP address and the port number.

For Client-Server communication,

  - Sockets are to be configured at the two ends to initiate a connection,
  - Listen for incoming messages
  - Send the responses at both ends
  - Establishing a bidirectional communication.

Socket Types

Datagram Socket

- A datagram is an independent, self-contained piece of information sent over a network whose arrival, arrival time, and content are not guaranteed. A datagram socket uses User Datagram Protocol (UDP) to facilitate the sending of datagrams (self-contained pieces of information) in an unreliable manner. Unreliable means that information sent via datagrams isn't guaranteed to make it to its destination.

Stream Socket:

- A stream socket, or connected socket, is a socket through which data can be transmitted continuously. A stream socket is more akin to a live network, in which the communication link is continuously active. A stream socket is a "connected" socket through which data is transferred continuously.

Socket in Python

```
sock_obj = socket.socket( socket_family, socket_type, protocol=0)
```

socket_family: - Defines family of protocols used as transport mechanism.

Either AF_UNIX, or

AF_INET (IP version 4 or IPv4).

socket_type: Defines the types of communication between the two end-points.

SOCK_STREAM (for connection-oriented protocols, e.g., TCP), or

SOCK_DGRAM (for connectionless protocols e.g. UDP).

protocol: We typically leave this field or set this field to zero.

Example:

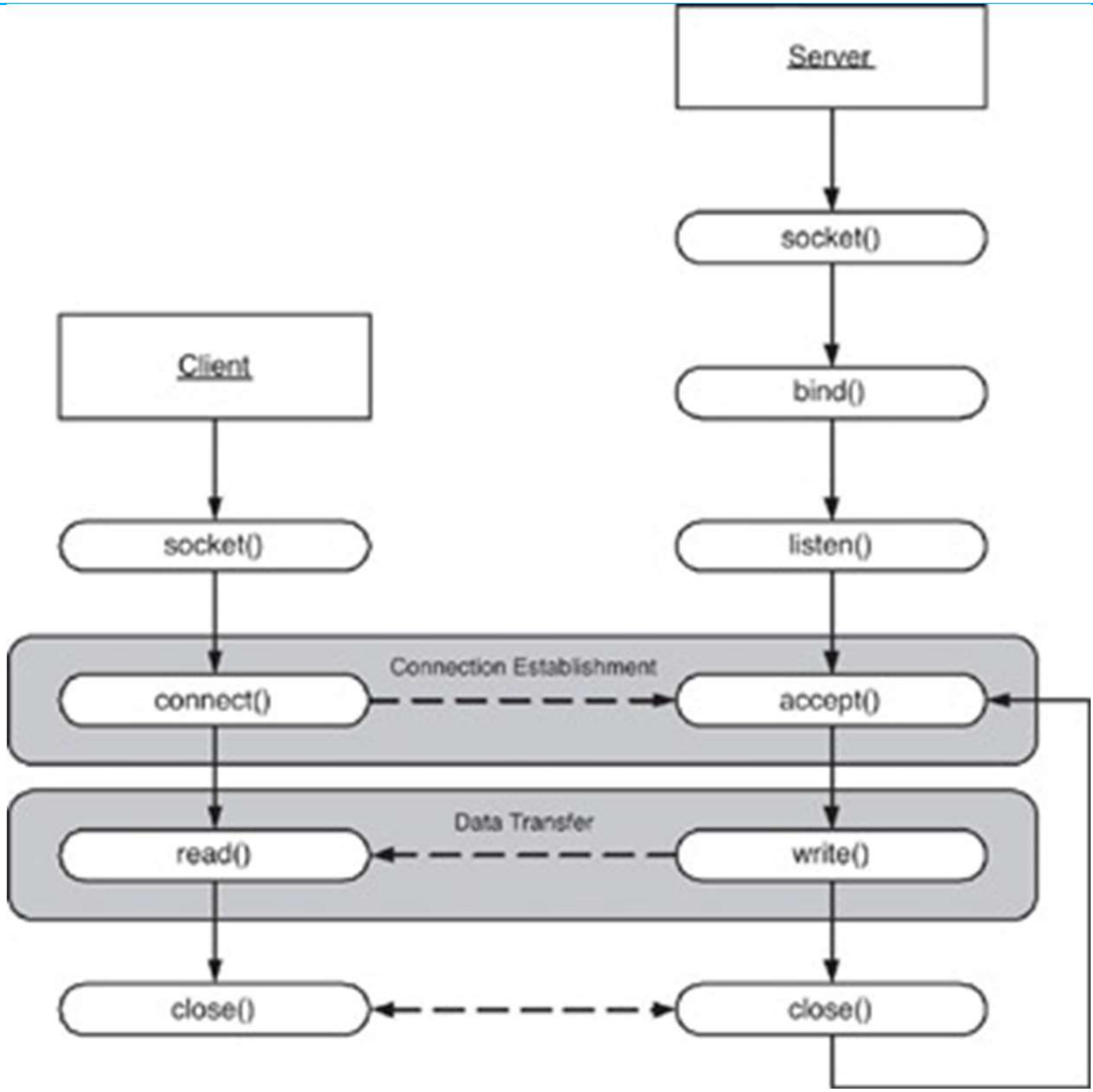
```
#Socket client example in python
import socket
#create an AF_INET, STREAM socket (TCP)
s = socket.socket(socket.AF_INET, socket.SOCK_STREAM)
print 'Socket Created'
```

Socket Creation

```
import socket
import sys
try:
        #create an AF_INET, STREAM socket (TCP)
        s = socket.socket(socket.AF_INET, socket.SOCK_STREAM)
except socket.error, msg:
        print 'Failed to create socket. Error code: ' + str(msg[0]) + ' , Error message : ' + msg[1]
        sys.exit();

print 'Socket Created'
```

Client/server symmetry in Sockets applications



Socket in Python

To create a socket, we must use socket.socket() function available in the Python socket module, which has the general syntax as follows:

S = socket.socket(socket_family, socket_type, protocol=0)

socket_family: This is either AF_UNIX or AF_INET. We are only going to talk about INET sockets in this tutorial, as they account for at least 99% of the sockets in use.

socket_type: This is either SOCK_STREAM or SOCK_DGRAM.

Protocol: This is usually left out, defaulting to 0.

Client Socket Methods

Following are some client socket methods:

connect( ) : To connect to a remote socket at an address. An address format(host, port) pair is used for AF_INET address family.

Socket in Python

Server Socket Methods

bind( ): This method binds the socket to an address. The format of address depends on socket family mentioned above(AF_INET).

listen(backlog) : This method listens for the connection made to the socket. The backlog is the maximum number of queued connections that must be listened before rejecting the connection.

accept( ) : This method is used to accept a connection. The socket must be bound to an address and listening for connections. The return value is a pair(conn, address) where conn is a new socket object which can be used to send and receive data on that connection, and address is the address bound to the socket on the other end of the connection.

General Socket in Python

sock_object.recv():

Use this method to receive messages at endpoints when the value of the protocol parameter is TCP.

sock_object.send():

Apply this method to send messages from endpoints in case the protocol is TCP.

sock_object.recvfrom():

Call this method to receive messages at endpoints if the protocol used is UDP.

sock_object.sendto():

Invoke this method to send messages from endpoints if the protocol parameter is UDP.

sock_object.gethostname():

This method returns hostname.

sock_object.close():

This method is used to close the socket. The remote endpoint will not receive data from this side.

Simple TCP Server

```
#!/usr/bin/python

#This is tcp_server.py script

import socket                          #line 1: Import socket module

s = socket.socket()                    #line 2: create a socket object
host = socket.gethostname()            #line 3: Get current machine name
port = 9999                            #line 4: Get port number for connection

s.bind((host,port))                    #line 5: bind with the address

print "Waiting for connection..."
s.listen(5)                            #line 6: listen for connections

while True:
    conn,addr = s.accept()             #line 7: connect and accept from client
    print 'Got Connection from', addr
    conn.send('Server Saying Hi')
    conn.close()                       #line 8: Close the connection
```

Simple TCP Client

```
#!/usr/bin/python

#This is tcp_client.py script

import socket

s = socket.socket()
host = socket.gethostname()        # Get current machine name
port = 9999                        # Client wants to connect to server's
                                   # port number 9999

s.connect((host,port))
print s.recv(1024)                 # 1024 is bufsize or max amount
                                   # of data to be received at once

s.close()
```

Simple UDP Server

```
#!usr/bin/python

import socket

sock = socket.socket(socket.AF_INET,socket.SOCK_DGRAM)      # For UDP

udp_host = socket.gethostname()              # Host IP
udp_port = 12345                             # specified port to connect

#print type(sock) ============> 'type' can be used to see type
                # of any variable ('sock' here)

sock.bind((udp_host,udp_port))

while True:
    print "Waiting for client..."
    data,addr = sock.recvfrom(1024)          #receive data from client
    print "Received Messages:",data," from",addr
```

Simple UDP Client

```
#!usr/bin/python

import socket

sock = socket.socket(socket.AF_INET,socket.SOCK_DGRAM)      # For UDP

udp_host = socket.gethostname()     # Host IP
udp_port = 12345                    # specified port to connect

msg = "Hello Python!"
print "UDP target IP:", udp_host
print "UDP target Port:", udp_port

sock.sendto(msg,(udp_host,udp_port))        # Sending message to UDP server
```

Symbolic Programming Paradigm

Introduction

Symbolic computation deals with the computation of mathematical objects symbolically. This means that the mathematical objects are represented exactly, not approximately, and mathematical expressions with unevaluated variables are left in symbolic form.

It Covers the following:

- As A calculator and symbols
- Algebraic Manipulations - Expand and Simplify
- Calculus – Limits, Differentiation, Series , Integration
- Equation Solving – Matrices

Calculator and Symbols

Rational - ½, or 5/2

```
>>import sympy as sym
>>a = sym.Rational(1, 2)
>>a
```
Answer will be 1/2

Constants like pi,e

```
>>sym.pi**2
```
Answer is pi**2

```
>>sym.pi.evalf()
```
Answer is 3.14159265358979

```
>> (sym.pi + sym.exp(1)).evalf()
```
Answer is 5.85987448204884

X AND Y

```
>> x = sym.Symbol('x')
>>y = sym.Symbol('y')
>>x + y + x – y
```
Answer is 2*x

Algebraic Manipulations

EXPAND (X+Y)3 = X+3X^2Y+3XY^2+Y**

```
>> sym.expand((x + y) ** 3)
```
Answer is x**3 + 3*x**2*y + 3*x*y**2 + y**3

```
>> 3 * x * y ** 2 + 3 * y * x ** 2 + x ** 3 + y ** 3
```
Answer is x**3 + 3*x**2*y + 3*x*y**2 + y**3

WITH TRIGNOMETRY LIKE SIN,COSINE

eg. COS(X+Y)= - SIN(X)*SIN(Y)+COS(X)*COS(Y)

```
>> sym.expand(sym.cos(x + y), trig=True)
```
Answer is -sin(x)*sin(y) + cos(x)*cos(y)

SIMPLIFY

(X+X*Y/X)=Y+1

```
>>sym.simplify((x + x * y) / x)
```
Answer is: y+1

Calculus

LIMITS compute the limit of

limit(function, variable, point)

limit(sin(x)/x , x, 0) =1

Differentiation

diff(func,var) eg diff(sin(x),x)=cos(x)

diff(func,var,n) eg

Series

series(expr,var)

series(cos(x),x) = 1-x/2+x/24+o(x)

Integration

Integrate(expr,var)

Integrate(sin(x),x) = -cos(x)

Example

Example:

```
In [23]: sym.expand(sym.cos(x + y), trig=True)
Out[23]: -sin(x)*sin(y) + cos(x)*cos(y)

In [24]: sym.limit(sym.sin(x) / x, x, 0)
Out[24]: 1

In [26]: sym.diff(sym.sin(x), x)
Out[26]: cos(x)

In [27]: sym.diff(sym.sin(2 * x), x)
Out[27]: 2*cos(2*x)

In [28]: sym.diff(sym.tan(x), x)
Out[28]: tan(x)**2 + 1

In [29]: sym.diff(sym.sin(2 * x), x, 1)
Out[29]: 2*cos(2*x)

In [30]: sym.diff(sym.sin(2 * x), x, 2)
Out[30]: -4*sin(2*x)

In [31]: sym.diff(sym.sin(2 * x), x, 3)
Out[31]: -8*cos(2*x)
```


```
In [31]: sym.diff(sym.sin(2 * x), x, 3)
Out[31]: -8*cos(2*x)
```


```
In [32]: sym.series(sym.cos(x), x)
Out[32]: 1 - x**2/2 + x**4/24 + O(x**6)

In [34]: sym.integrate(6 * x ** 5, x)
Out[34]: x**6

In [35]: sym.integrate(sym.sin(x), x)
Out[35]: -cos(x)

In [36]: sym.integrate(sym.log(x), x)
Out[36]: x*log(x) - x

In [37]: sym.integrate(2 * x + sym.sinh(x), x)
Out[37]: x**2 + cosh(x)
```


```
In [37]: sym.integrate(2 * x + sym.sinh(x), x)
Out[37]: x**2 + cosh(x)
```


```
In [38]: sym.integrate(sym.exp(-x ** 2) * sym.erf(x), x)
Out[38]: sqrt(pi)*erf(x)**2/4

In [39]: sym.integrate(x**3, (x, -1, 1))
Out[39]: 0

In [40]: sym.integrate(sym.sin(x), (x, 0, sym.pi / 2))
Out[40]: 1
```

Example

Example:

```
In [43]: sym.solveset(x ** 4 - 1, x)
Out[43]: {-1, 1, -I, I}

In [44]: sym.solveset(sym.exp(x) + 1, x)
Out[44]: ImageSet(Lambda(_n, I*(2*_n*pi + pi)), S.Integers)

In [46]: solution = sym.solve((x + 5 * y - 2, -3 * x + 6 * y - 15), (x, y))
         solution[x], solution[y]
Out[46]: (-3, 1)

In [47]: f = x ** 4 - 3 * x ** 2 + 1
         sym.factor(f)
Out[47]: (x**2 - x - 1)*(x**2 + x - 1)

In [48]: sym.satisfiable(x & y)
Out[48]: {x: True, y: True}
```

```
In [49]: sym.Matrix([[1, 0], [0, 1]])
Out[49]: Matrix([
         [1, 0],
         [0, 1]])

In [51]: x, y = sym.symbols('x, y')
         A = sym.Matrix([[1, x], [y, 1]])
         A
Out[51]: Matrix([
         [1, x],
         [y, 1]])

In [52]: A**2
Out[52]: Matrix([
         [x*y + 1,     2*x],
         [    2*y, x*y + 1]])
```

Equation Solving

solveset()

solveset(x ** 4 - 1, x) ={-1,1,-I,I}

Matrices

A={[1,2][2,1]} find A**2

Automata Based Programming Paradigm

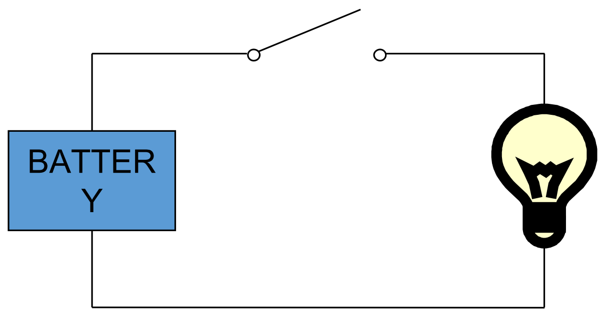
Introduction

Automata-based programming is a programming paradigm in which the program or its part is thought of as a model of a finite state machine or any other formal automation.

What is Automata Theory?

- Automata theory is the study of abstract computational devices
- Abstract devices are (simplified) models of real computations
- Computations happen everywhere: On your laptop, on your cell phone, in nature, …

Example:



input: switch

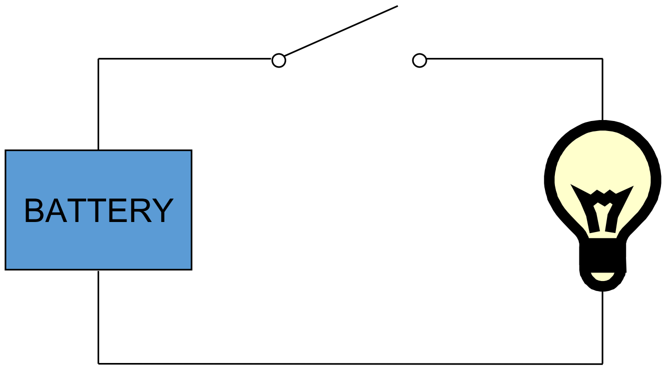
output: light bulb

actions: flip switch

states: on, off

Simple Computer

Example:

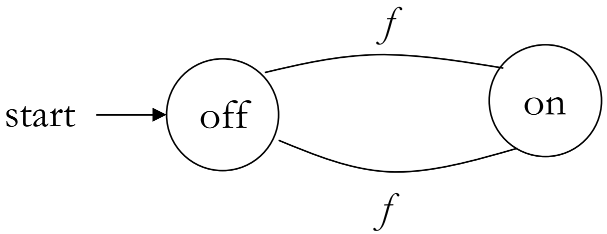


input: switch

output: light bulb

actions: flip switch

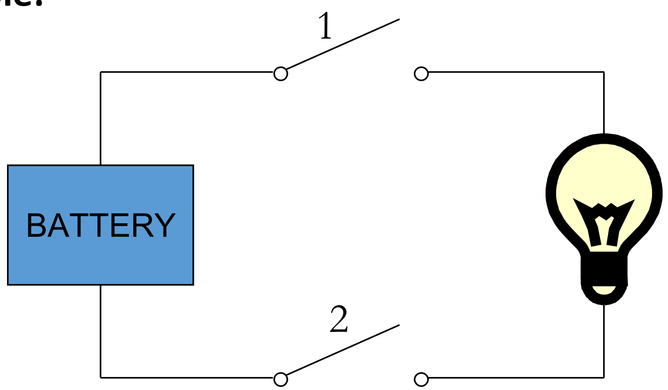
states: on, off



bulb is on if and only if there was an odd number of flips

Another “computer”

Example:

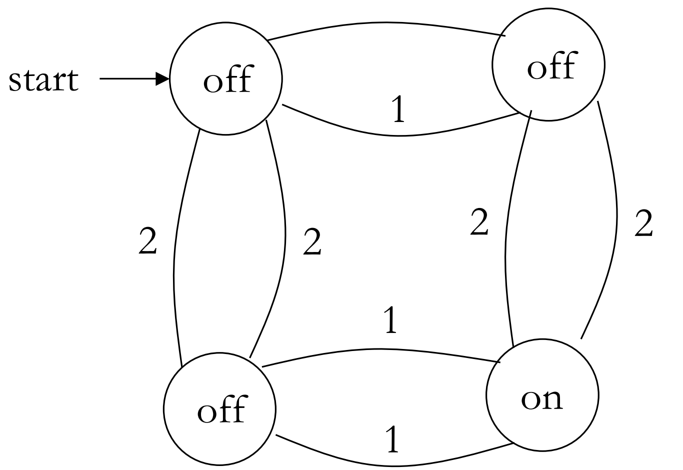


inputs: switches 1 and 2

actions: 1 for “flip switch 1”

actions: 2 for “flip switch 2”

states: on, off



bulb is on if and only if both switches were flipped an odd number of times

Types of Automata

finite automata	Devices with a finite amount of memory. Used to model "small" computers.
push-down automata	Devices with infinite memory that can be accessed in a restricted way. Used to model parsers, etc.
Turing Machines	Devices with infinite memory. Used to model any computer.

Alphabets and strings

A common way to talk about words, number, pairs of words, etc. is by representing them as strings

To define strings, we start with an alphabet

An alphabet is a finite set of symbols.

Examples:

$\Sigma_1 = \{a, b, c, d, \ldots, z\}$: the set of letters in English

$\Sigma_2 = \{0, 1, \ldots, 9\}$: the set of (base 10) digits

$\Sigma_3 = \{a, b, \ldots, z, \#\}$: the set of letters plus the special symbol #

$\Sigma_4 = \{\ (,\ )\ \}$: the set of open and closed brackets

Strings

A string over alphabet Σ is a finite sequence of symbols in Σ.

The empty string will be denoted by e

Examples:

abfbz is a string over $\Sigma_1 = \{a, b, c, d, \ldots, z\}$

9021 is a string over $\Sigma_2 = \{0, 1, \ldots, 9\}$

ab#bc is a string over $\Sigma_3 = \{a, b, \ldots, z, \#\}$

))()(() is a string over $\Sigma_4 = \{ (,) \}$

Languages

A language is a set of strings over an alphabet.

Languages can be used to describe problems with “yes/no” answers, for example:

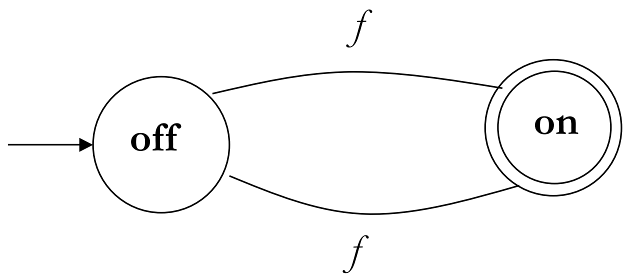
$L_1 =$ The set of all strings over Σ_1 that contain the substring “SRM”

$L_2 =$ The set of all strings over Σ_2 that are divisible by 7 $= \{7, 14, 21, \ldots\}$

$L_3 =$ The set of all strings of the form $s\#s$ where s is any string over $\{a, b, \ldots, z\}$

$L_4 =$ The set of all strings over Σ_4 where every (can be matched with a subsequent)

Finite Automata



There are states off and on, the automaton starts in off and tries to reach the “good state” on

What sequences of fs lead to the good state?

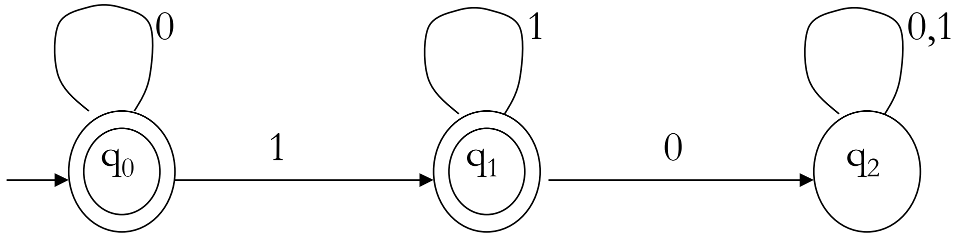
Answer: {f, fff, fffff, …} = {f n: n is odd}

This is an example of a deterministic finite automaton over alphabet {f}

Deterministic finite automata

- A deterministic finite automaton (DFA) is a 5-tuple $(\mathcal{Q}, \Sigma, \delta, q_0, F)$ where
  - $\mathcal{Q}$ is a finite set of states
  - Σ is an alphabet
  - $\delta: \mathcal{Q} \times \Sigma \to \mathcal{Q}$ is a transition function
  - $q_0 \in \mathcal{Q}$ is the initial state
  - $F \subseteq \mathcal{Q}$ is a set of accepting states (or final states).
- In diagrams, the accepting states will be denoted by double loops

Example



transition function δ:

states	inputs: 0	1
q_0	q_0	q_1
q_1	q_2	q_1
q_2	q_2	q_2

alphabet $\Sigma = \{0, 1\}$

start state $Q = \{q_0, q_1, q_2\}$

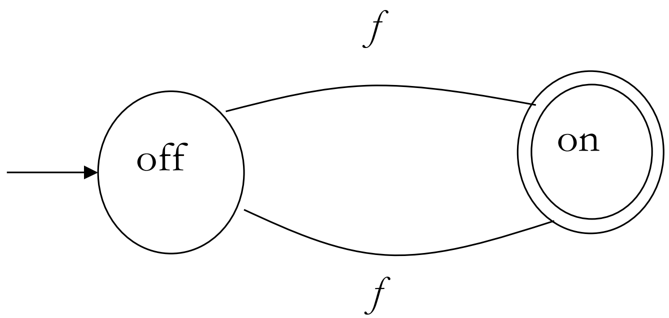
initial state q_0

accepting states $F = \{q_0, q_1\}$

Language of a DFA

The language of a DFA $(Q, \Sigma, \delta, q_0, F)$ is the set of all strings over Σ that, starting from q_0 and following the transitions as the string is read left to right, will reach some accepting state.

M:

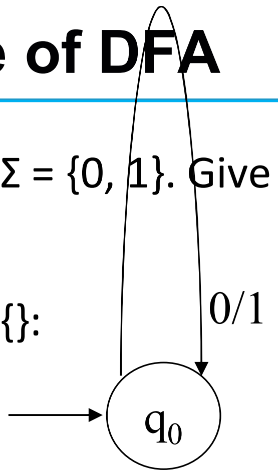


- Language of M is $\{f, fff, fffff, \ldots\} = \{f^n: n \text{ is odd}\}$

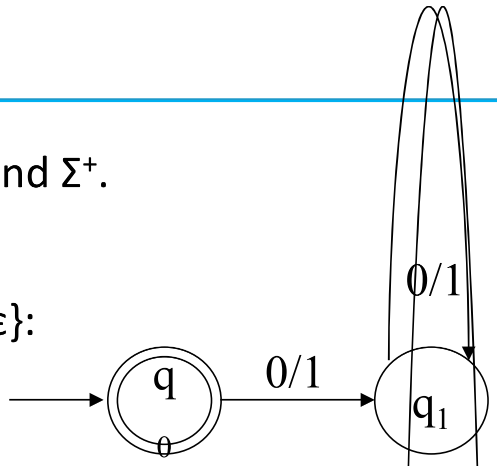
Example of DFA

1. Let $\Sigma = \{0, 1\}$. Give DFAs for $\{\}$, $\{\varepsilon\}$, Σ^*, and Σ^+.

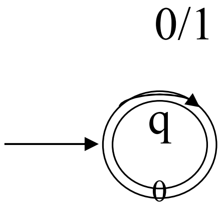
For $\{\}$:



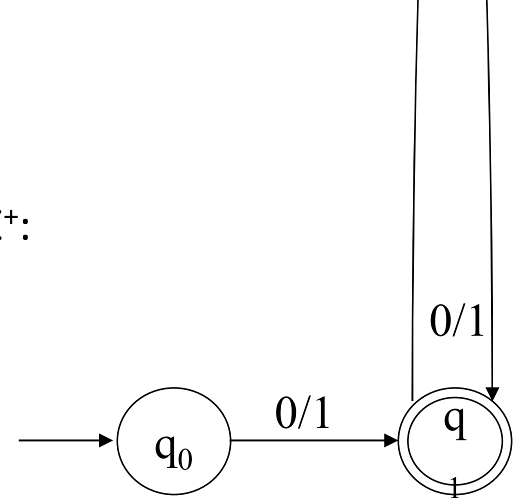
For $\{\varepsilon\}$:



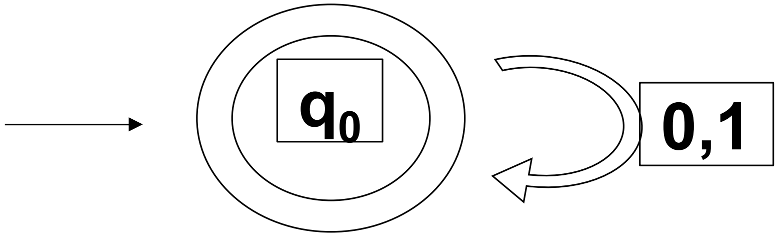
For Σ^*:



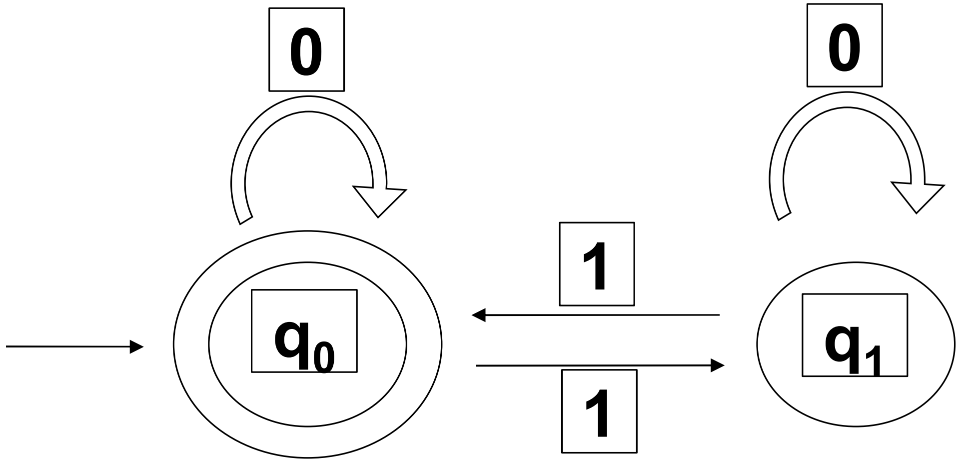
For Σ^+:



Example of DFA



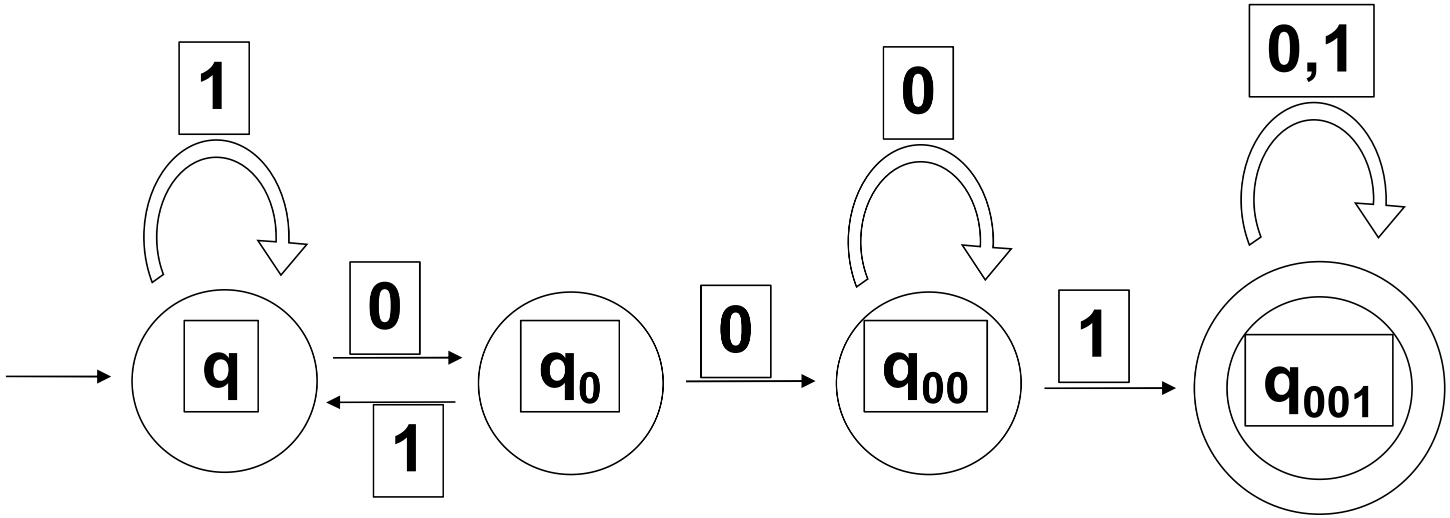
L(M) = {0,1}*



L(M) = { w | w has an even number of 1s}

Example of DFA

Build an automaton that accepts all and only those strings that contain 001



Example of DFA using Python

```
from automata.fa.dfa import DFA
# DFA which matches all binary strings ending in an odd number of '1's
dfa = DFA(
   states={'q0', 'q1', 'q2'},
   input_symbols={'0', '1'},
   transitions={
      'q0': {'0': 'q0', '1': 'q1'},
      'q1': {'0': 'q0', '1': 'q2'},
      'q2': {'0': 'q2', '1': 'q1'}
   },
   initial_state='q0',
   final_states={'q1'}
)
dfa.read_input('01')   # answer is  'q1'
dfa.read_input('011')  # answer is error
print(dfa.read_input_stepwise('011'))
Answer # yields:
# 'q0'     # 'q0'     # 'q1'
# 'q2'     # 'q1'
```

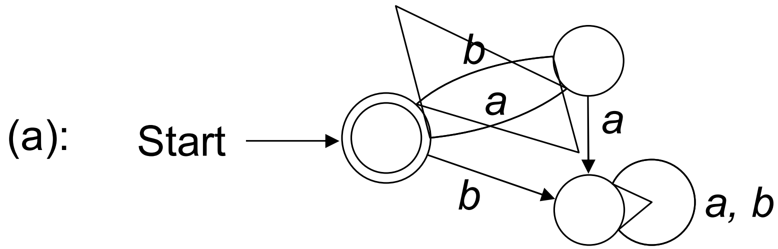
```
if dfa.accepts_input('011'):
   print('accepted')
else:
   print('rejected')
```

Questions for DFA

Find an DFA for each of the following languages over the alphabet $\{a, b\}$.

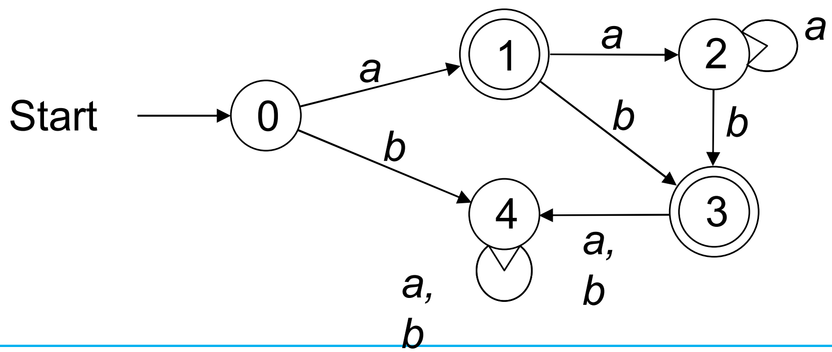
(a) $\{(ab)^n \mid n \in \mathrm{N}\}$, which has regular expression $(ab)^*$.

Solution
:



b) Find a DFA for the language of $a + aa^*b$.

Solution
:



Questions for DFA

c) A DFA that accepts all strings that contain 010 or do not contain 0.

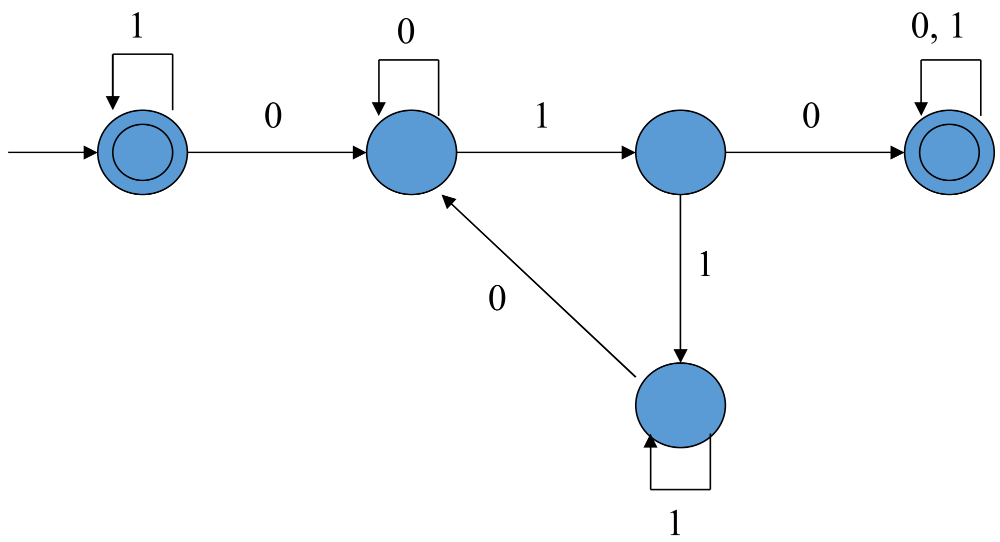
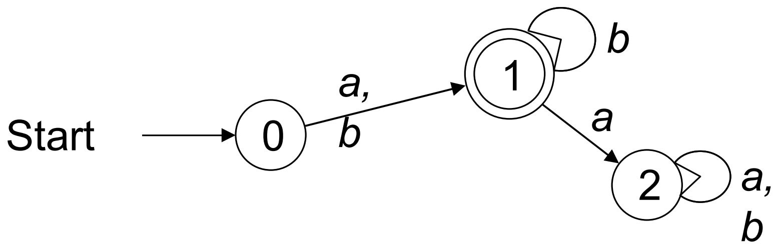


Table Representation of a DFA

A DFA over A can be represented by a transition function T : States X A -> States, where T(i, a) is the state reached from state i along the edge labelled a, and we mark the start and final states. For example, the following figures show a DFA and its transition table.



	T	a	b
start	0	1	1
final	1	2	1
	2	2	2

Sample Exercises - DFA

1. Write a automata code for the Language that accepts all and only those strings that contain 001
2. Write a automata code for L(M) ={ w | w has an even number of 1s}
3. Write a automata code for L(M) ={0,1}*
4. Write a automata code for L(M)=$a + aa^*b$.
5. Write a automata code for L(M)={$(ab)^n$ | $n \in$ N}
6. Write a automata code for Let $\Sigma = \{0, 1\}$.

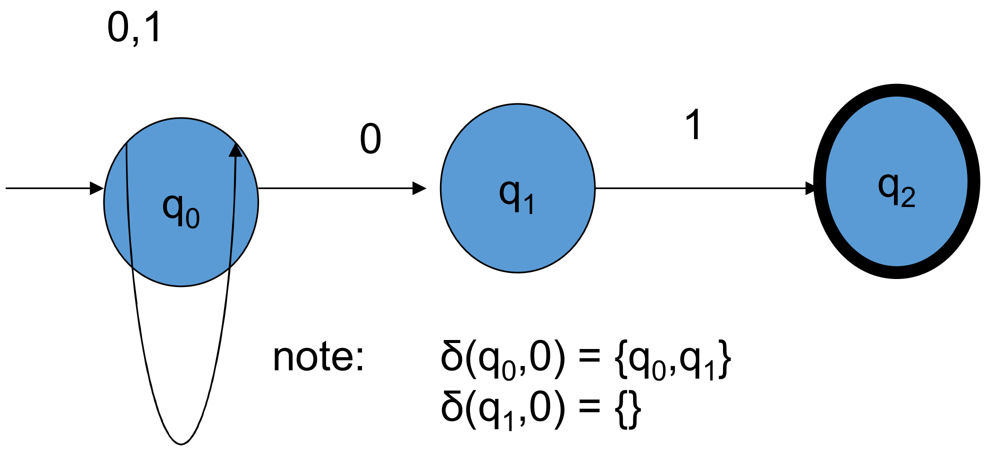
   Given DFAs for {}, {ε}, Σ^*, and Σ^+.

NDFA

- A nondeterministic finite automaton M is a five-tuple $M = (Q, \Sigma, \delta, q_0, F)$, where:
  - Q is a finite set of states of M
  - Σ is the finite input alphabet of M
  - $\delta: Q \times \Sigma \rightarrow$ power set of Q, is the state transition function mapping a state-symbol pair to a subset of Q
  - q_0 is the start state of M
  - $F \subseteq Q$ is the set of accepting states or final states of M

Example NDFA

- NFA that recognizes the language of strings that end in 01



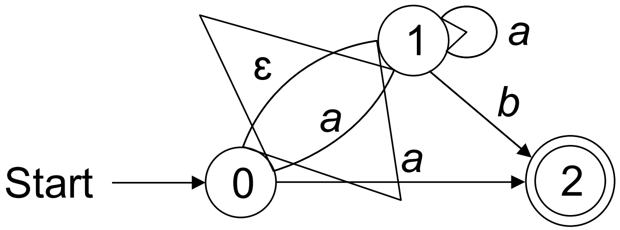
note: $\delta(q_0,0) = \{q_0,q_1\}$
$\delta(q_1,0) = \{\}$

Exercise: Draw the complete transition table for this NFA

NDFA

A nondeterministic finite automaton (NFA) over an alphabet A is similar to a DFA except that epislon-edges are allowed, there is no requirement to emit edges from a state, and multiple edges with the same letter can be emitted from a state.

Example. The following NFA recognizes the language of a + aa*b + a*b.

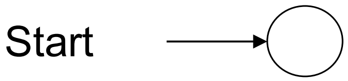


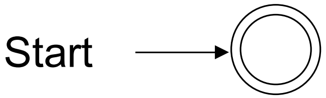
	T	a	b	e
start	0	{1, 2}	∅	{1}
	1	{1}	{2}	∅
final	2	∅	∅	∅

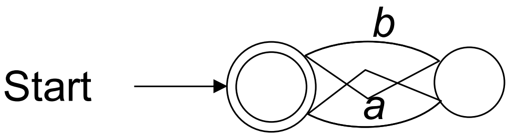
Table representation of NFA

An NFA over A can be represented by a function T : States × A ∪ {L} → power(States), where T(i, a) is the set of states reached from state i along the edge labeled a, and we mark the start and final states. The following figure shows the table for the preceding NFA.

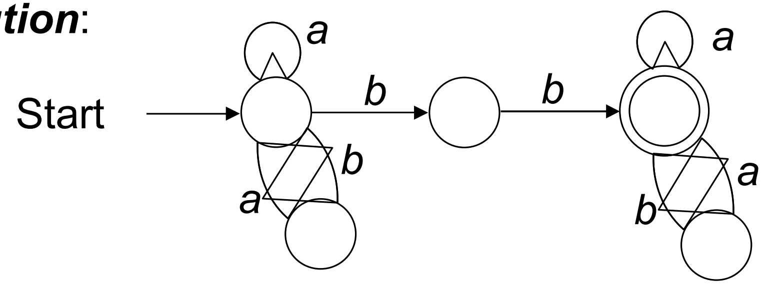
Examples

Solutions: (a): 

(b)
: 

(c): 

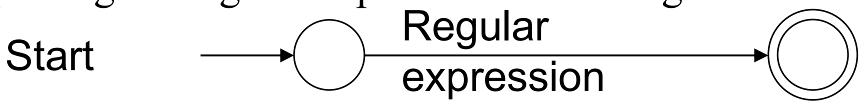
Find an NFA to recognize the language (a + ba)*bb(a + ab)*.

A solution: 

Examples

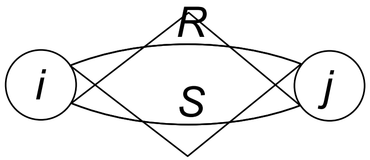
Algorithm: *Transform a Regular Expression into a Finite Automaton*

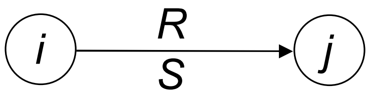
Start by placing the regular expression on the edge between a start and final state:



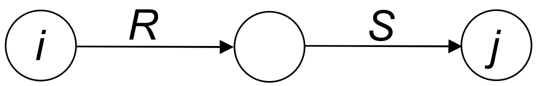
Apply the following rules to obtain a finite automaton after erasing any ∅-edges.

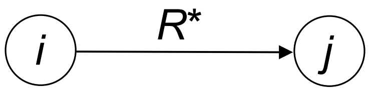
transforms to



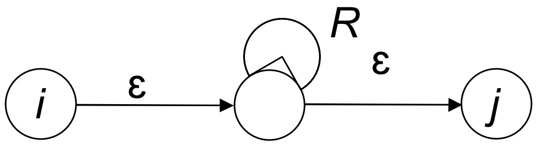


transforms to



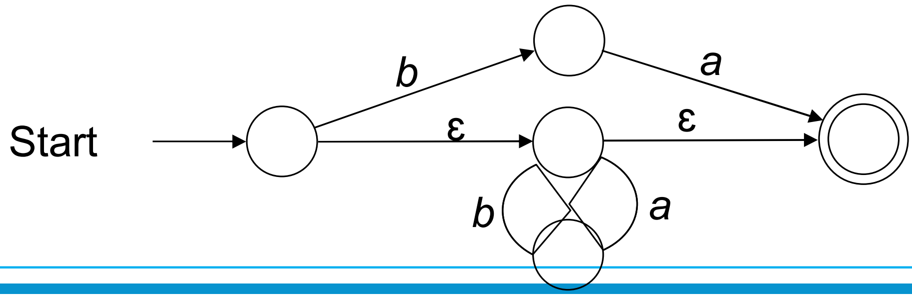


transforms to



Quiz. Use the algorithm to construct a finite automaton for (*ab*)* + *ba*.

Answer:



Example of NFA using Python

```
from automata.fa.nfa import NFA
# NFA which matches strings beginning with 'a', ending with 'a', and containing
# no consecutive 'b's
nfa = NFA(
   states={'q0', 'q1', 'q2'},
   input_symbols={'a', 'b'},
   transitions={
      'q0': {'a': {'q1'}},
      # Use '' as the key name for empty string (lambda/epsilon) transitions
      'q1': {'a': {'q1'}, '': {'q2'}},
      'q2': {'b': {'q0'}}
   },
   initial_state='q0',
   final_states={'q1'}
)
```

```
nfa.read_input('aba')
ANSWER :{'q1', 'q2'}

nfa.read_input('abba')
ANSWER: ERROR


nfa.read_input_stepwise('aba')

if nfa.accepts_input('aba'):
   print('accepted')
else:
   print('rejected')
ANSWER: ACCEPTED
nfa.validate()
ANSWR: TRUE
```

Sample Exercises - NFA

1. Write a automata code for the Language that accepts all end with 01
2. Write a automata code for L(M)= $a + aa^*b + a^*b$.
3. Write a automata code for Let $\Sigma = \{0, 1\}$.

   Given NFAs for $\{\}$, $\{\varepsilon\}$, $\{(ab)^n \mid n \in N\}$, which has regular expression $(ab)^*$.