Unit V
Automata Based Programming Paradigm

Topics Covered

- Automata Based Programming Paradigm
- Finite State Machine, deterministic finite automation (dfa), nfa
- State transitions using python-automaton Initial state, destination state, event (transition)
- Other languages: Forth, Ragel, SCXML
- Demo: Automata Based Programming in Python

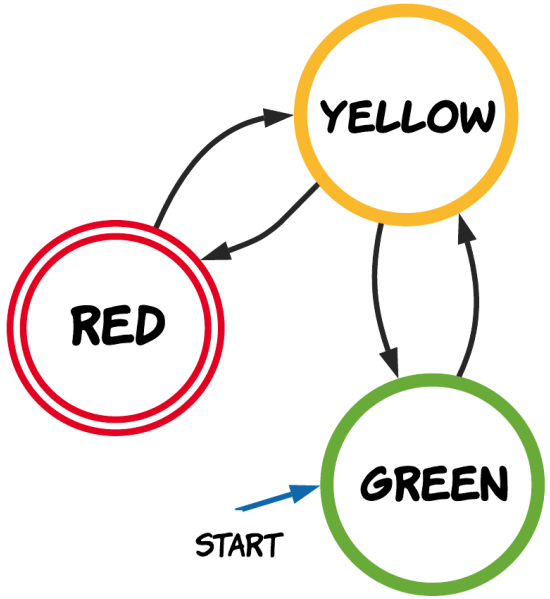
Reference Book :“A Review of Programming Paradigms throughout the History” by Elad Shalom
https://pypi.org/project/automata-lib

Introduction

- In our day-to-day life, we encounter various state-based systems.
- These systems produce an output based on the series of inputs given to them.
- Some instances include traffic lights, elevators, vending machines.
- Such, systems can be expressed logically as ***finite automata***.

Introduction

- One of the possible state diagrams for a traffic light



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, …

Types of Automata

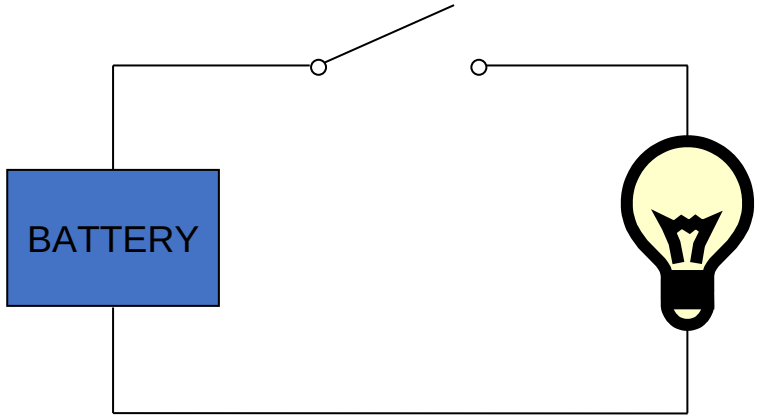
finite automata	Devices with a finite amount of memory. Used to model “small” computers.
push-down automata	Devices with infinite memory that can be accessed in a restricted way. Used to model parsers, etc.
Turing Machines	Devices with infinite memory. Used to model any computer.

What are Finite Automata?

- Finite automata (also called finite-state automata) is an abstract machine, which consists of a finite number of states.
- They can be in only one of these states at a time.
- State change occurs when input is given.
- Depending on the present state and input, the machine transitions to the new state.

Simple Computer

Example:

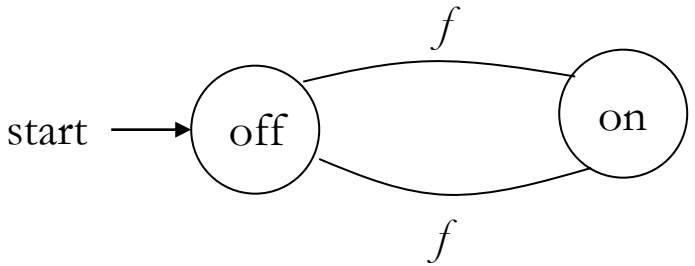


input: switch

output: light bulb

actions: flip switch

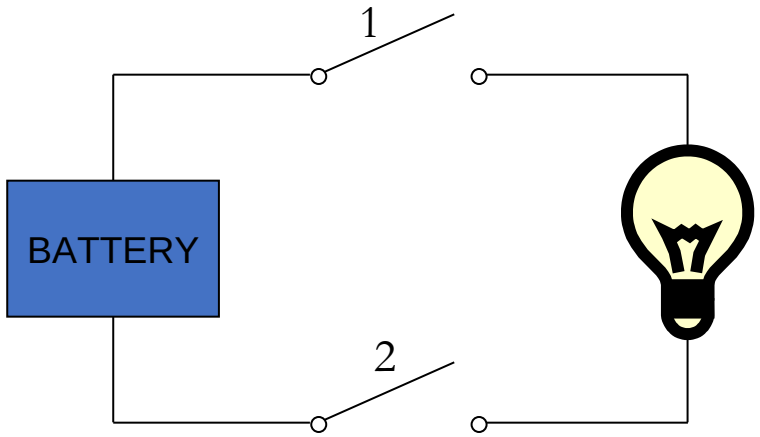
states: on, off



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

Another “computer”

Example:

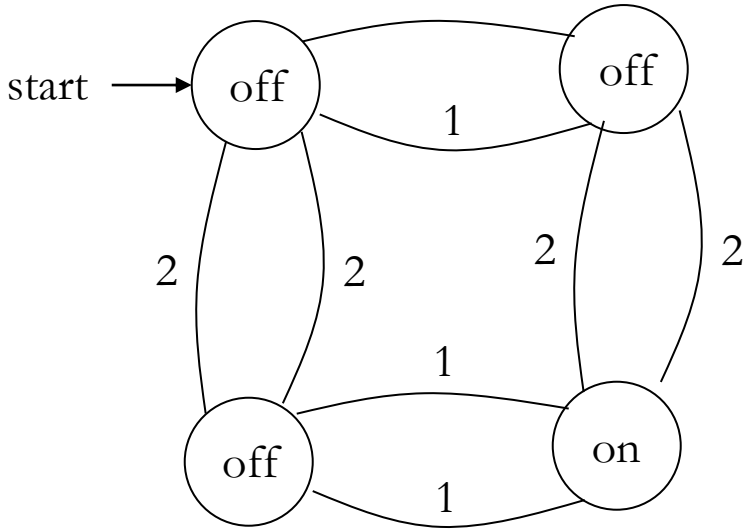


inputs: switches 1 and 2

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

Finite Automata

- Finite automata are **formally defined** as a 5-tuple (Q, Σ, δ, $q0$, F), where,
  - Q is a finite set of **states**
  - Σ is a finite set of input symbols (called **alphabets**)
  - δ is a **transition** function
  - $q0$ is the **start state**
  - F is a finite set of **accepting states**
- we can represent a FA graphically, with nodes for states, and arcs for transitions.

Finite Automata

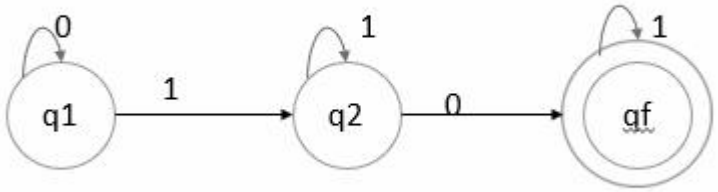
- **Examples**
  - 4-state FA to recognize words with 3 x's
  - 3-state FA to recognize Pascal variable names (letter followed by one or more letters or digits)
  - 4-state FA to recognize binary strings that end with 111

Graphical Representation of a FA

- A FA is represented by digraphs called Transition Diagram
- It is a directed graph associated with the vertices of the graph corresponding to the state of finite automata.
- The vertices represent the states.
- The arcs labeled with an input alphabet show the transitions.
- The initial state is denoted by an empty single incoming arc.
- The final state is indicated by double circles.

Transition Diagram…

- An example of transition diagram is given below



- Here,
  - {0,1}: Inputs
  - q1: Initial state
  - q2: Intermediate state
  - qf: Final state

Finite Automata...

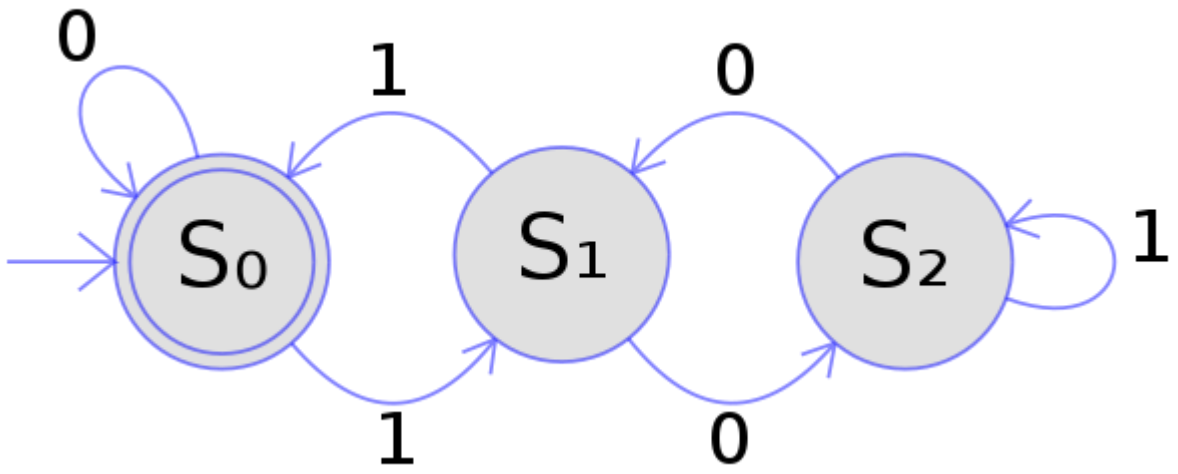
- A Finite Automata is a simple machine that recognizes patterns in the input string.
- If the input string contains the pattern defined by FA it accepts the string otherwise rejects it.
- 2 types:
  - Deterministic finite automata (DFA)
  - Non-deterministic finite automata (NDFA or NFA)

Finite Automata…

- The basic difference between DFA & NFA is that:
- For a particular input string, a machine can go to only 1 state in DFA but a machine can go to multiple states in NFA.
- Null transitions are allowed in NFA but not in DFA.
- Therefore, **all DFA are NFA but all NFA are not DFA**.

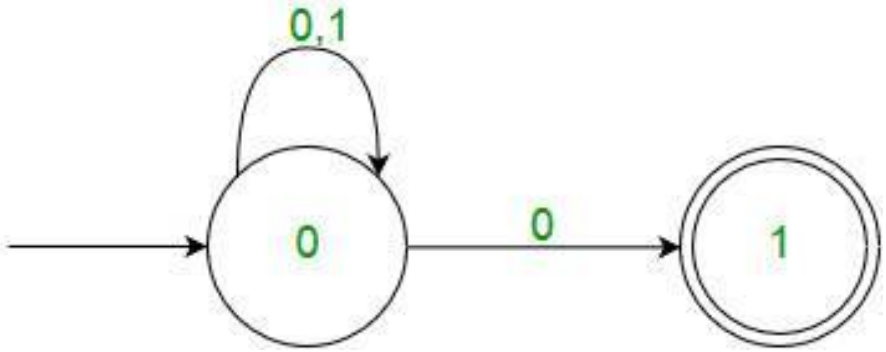
Example DFA

- DFA which accepts binary numbers which are multiples of 3

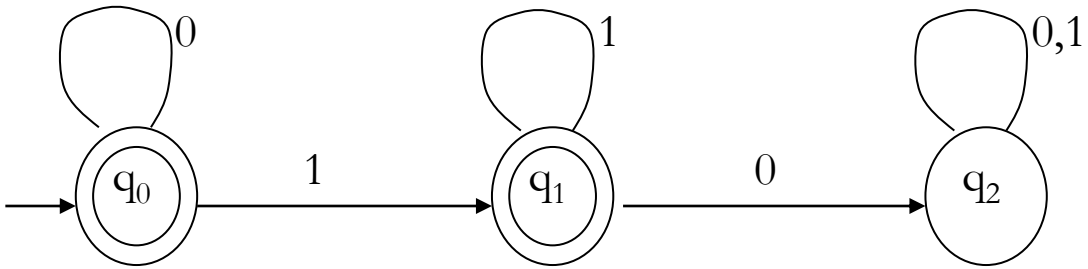


Example NFA

- NFA with Σ = {0, 1} accepts all strings ending with 0.



Example



alphabet $\Sigma = \{0, 1\}$
start state $Q = \{q_0, q_1, q_2\}$
initial state q_0
accepting states $F = \{q_0, q_1\}$

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

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 $= \{7, 14, 21, \ldots\}$

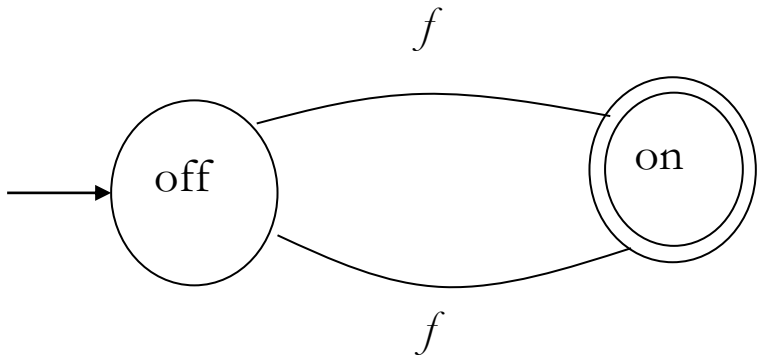
$L_3 =$ The set of all strings of the form s#s where s is any string over $\{a, b, \ldots, z\}$

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

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

Regular Expressions

- The languages which are accepted by finite automata are called **regular languages**.
- Regular expressions are patterns that are used to describe regular languages.
- Example
  1. language L over input alphabets ∑ = {a, b} such that L is the set of all strings starting with ‘aba’.

     Regular expression for the given language = aba(a + b)*
  2. language L over input alphabets ∑ = {0, 1} such that L is the set of all strings starting with ’00’.

     Regular expression for the given language = 00(0 + 1)*

Steps To Construct DFA

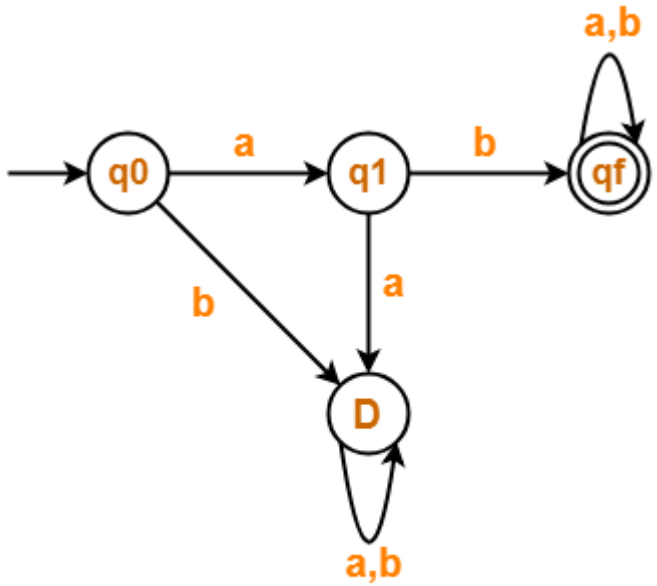
1. Determine the minimum number of states required in the DFA.
   - Use the following rule to determine the minimum number of states
     - Calculate the length of substring.
     - All strings starting with ‘n’ length substring will always require minimum (n+2) states in the DFA.
2. Draw those states.
3. Decide the strings for which DFA will be constructed.
4. Construct a DFA for the strings decided in previous step.

Example

- Draw a DFA for the language accepting strings starting with ‘ab’ over input alphabets ∑ = {a, b}
- Step 1:
  - All strings of the language starts with substring “ab”.
  - So, length of substring = 2.
  - Thus, Minimum number of states required in the DFA = 2 + 2 = 4.
  - It suggests that minimized DFA will have 4 states.
- Step 2: We will construct DFA for the following strings
  - ab
  - aba
  - abab

Example…

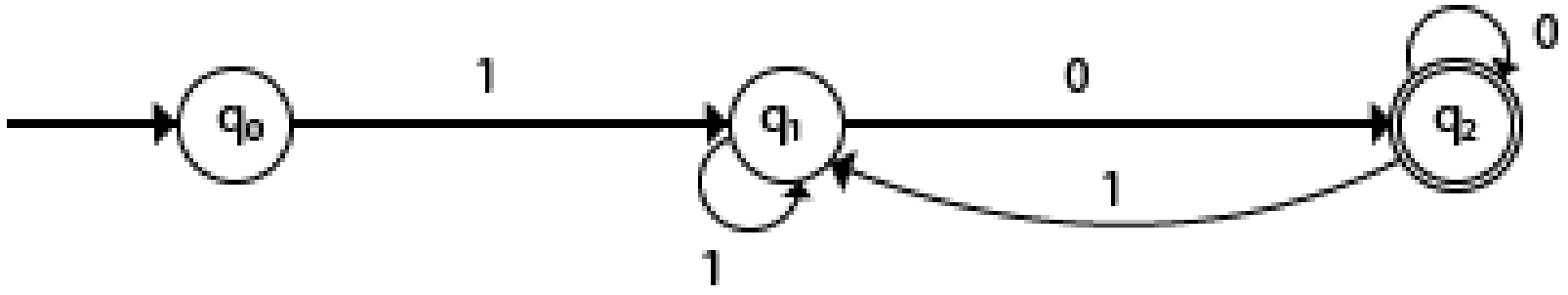
- The required DFA is given below



DFA

Examples of DFA

- Example 1:Design a FA with ∑ = {0, 1} accepts those string which starts with 1 and ends with 0.
- **Solution:**



- Example2: Design a FA with ∑ = {0, 1} accepts the only input 101.

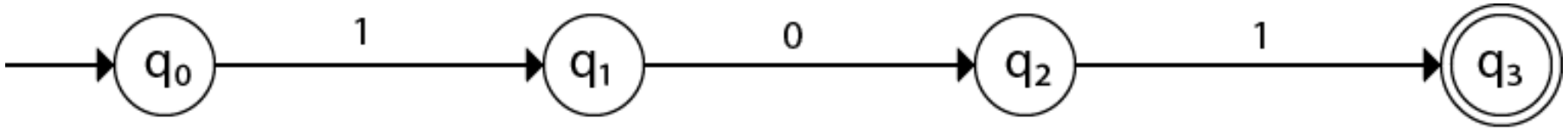
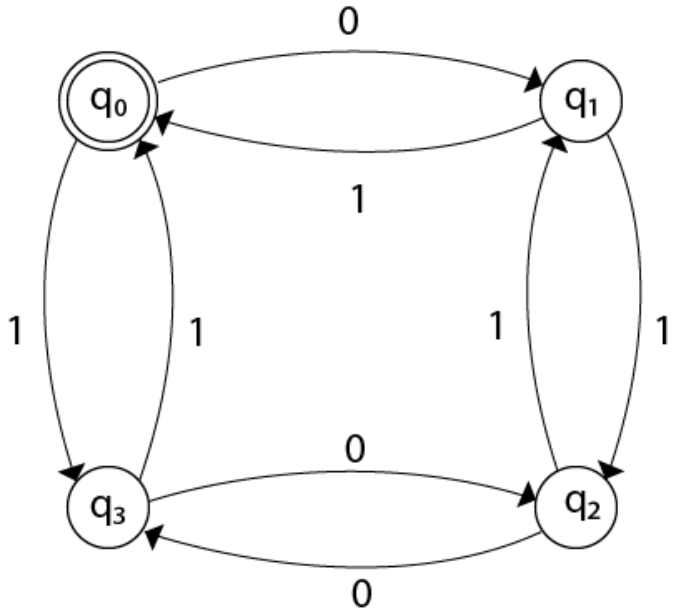


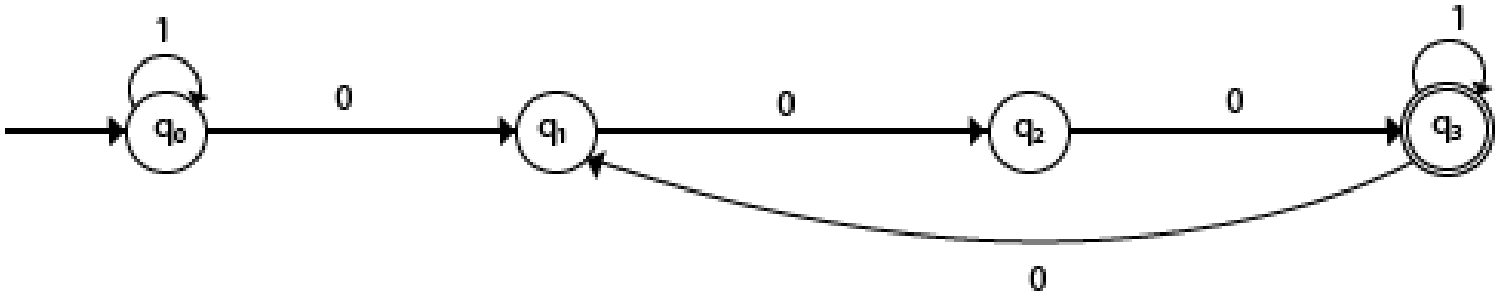
Fig: FA

- Example 3: Design FA with ∑ = {0, 1} accepts even number of 0's and even number of 1's.

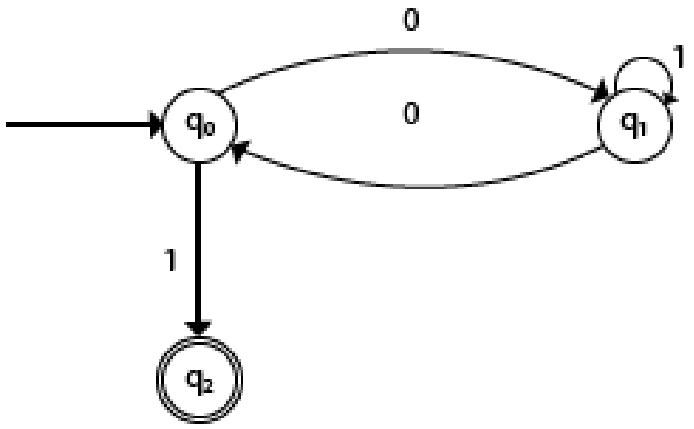


q0: state of even number of 0's and even number of 1's.
q1: state of odd number of 0's and even number of 1's.
q2: state of odd number of 0's and odd number of 1's.
q3: state of even number of 0's and odd number of 1's.

- Example 4: Design FA with ∑ = {0, 1} accepts the set of all strings with three consecutive 0's.
- **Solution:**
- The strings that will be generated for this particular languages are 000, 0001, 1000, 10001, ... in which 0 always appears in a clump of 3. The transition graph is as follows:



- Example 5: Design a FA with ∑ = {0, 1} accepts the strings with an even number of 0's followed by single 1.
- **Solution:**



Automata-based Programming Paradigm

- **Automata-based programming** is a computer programming paradigm that treats the program or part of it is as finite automata.
- Each automaton can take one "step" at a time, and the execution of the program is broken down into individual steps.
- The steps communicate with each other by changing the value of a variable, representing the "state."
- The control flow of the program is determined by the variable value.

Automata-based Programming Paradigm…

- The "state" variable can be a simple enum data type, but more complex data structures may be used.
- A common technique is to create a **state transition table**
  - a two-dimensional array comprising rows representing every possible state, and columns representing input parameter.
  - The table value where the row and column meet is the next state the machine should transition to if both conditions are met.

Automata-based Programming Paradigm…

- The following properties are key indicators for automata-based programming:
  - The time period of the program's execution is clearly separated down to the automaton steps. Each step is effectively an execution of a code section which has a single entry point. That section might be divided down to subsections to be executed depending on different states.
  - Any communication between the automaton steps is only possible via the explicitly noted set of variables named the automaton state.
    - Between any two steps, the program cannot have implicit components of its state, such as local variables' values, return addresses, the current instruction pointer, etc.
    - That is, the state of the whole program, taken at any two moments of entering an automaton step, can only differ in the values of the variables being considered as the automaton state.

Use of automata

- Widely used in lexical and syntactic analyses.
- The notions of states and state machines are often used in the field of formal specification.
- To describe semantics of some programming languages (Refal language).

Questions for DFA

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

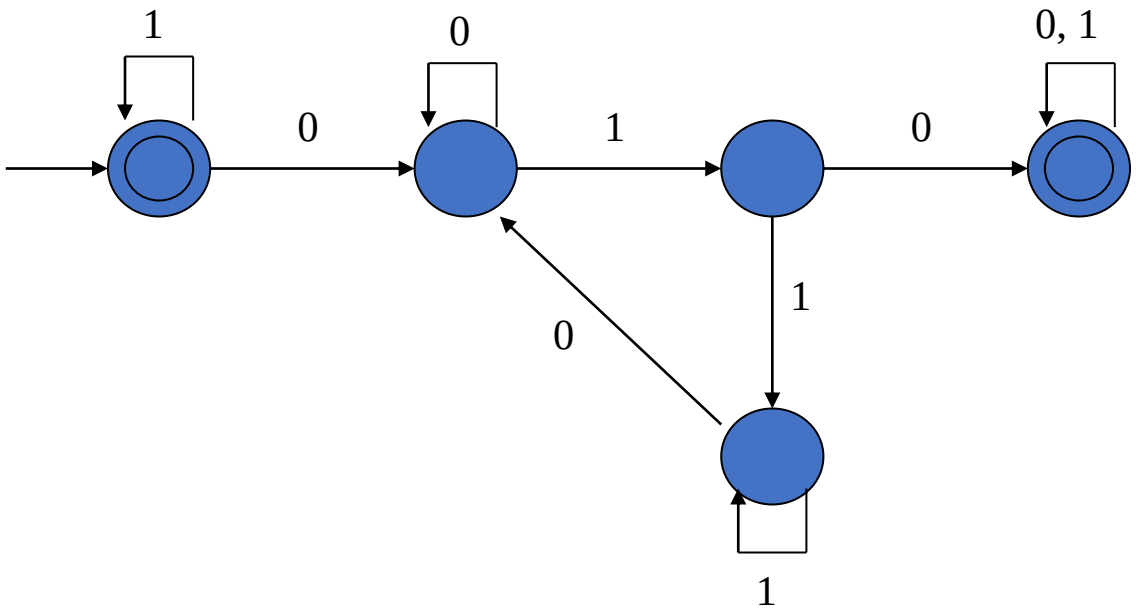
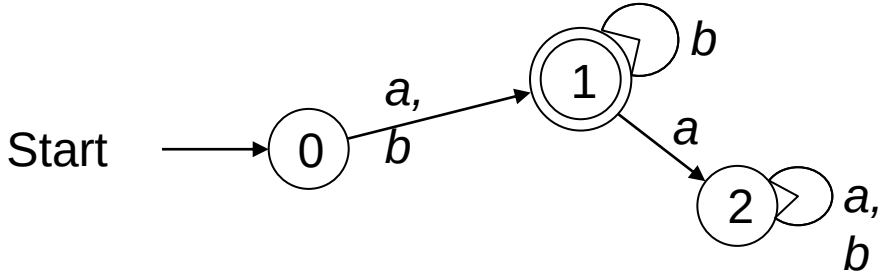


Table Representation of a DFA

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.



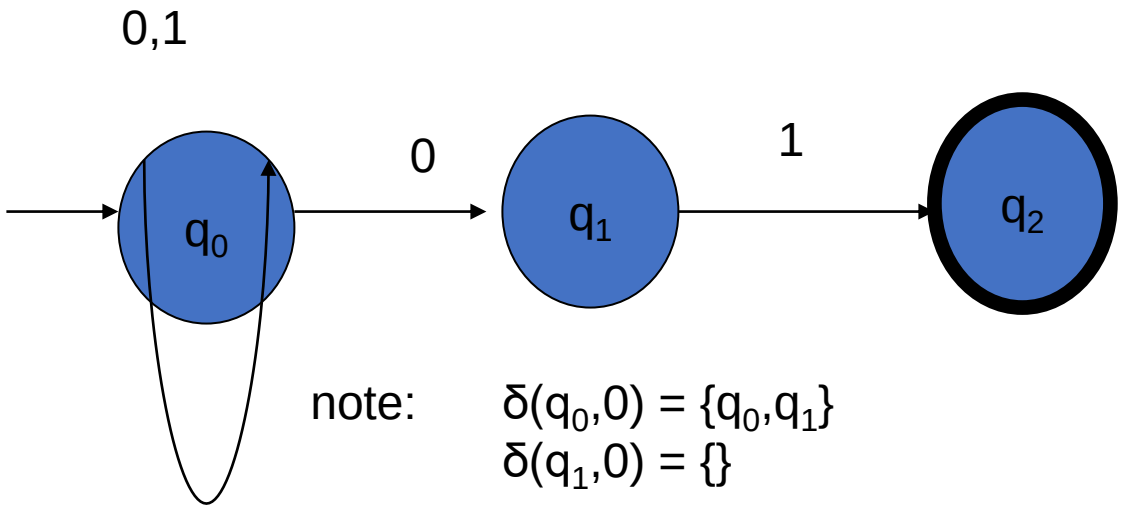
	T	a	b
start	0	1	1
final	1	2	1
	2	2	2

NDFA

- A nondeterministic finite automaton M is a five-tuple M = (Q, Σ, δ, q_0, F), where:
  - Q is a finite set of states of M
  - Σ is the finite input alphabet of M
  - δ: Q × Σ → 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 ⊆ 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



Exercise: Draw the complete transition table for this NFA

Automata Based Programming in Python

Automata Programming in Python

- The steps required to perform automata programming are :
  - Firstly, determine the total numbers of states and inputs
  - Secondly, transitions for each state
  - Thirdly, set the final state

- Example
  - L={W| W starts with 1 and ends with 0}
  - Total number of states=4 {q0,q1,q2,q3}
  - inputs= 0 and 1

Automata Library

- Automata is a Python 3 library which implements the structures and algorithms for finite automata, pushdown automata, and Turing machines.
- Install the latest version of Automata via pip:

  pip install automata-lib
- **class FA(Automaton, metaclass=ABCMeta)**
  - The FA class is an abstract base class from which all finite automata inherit.

    from automata.fa.fa import FA

class DFA(FA)

- The DFA class is a subclass of FA and represents a deterministic finite automaton.
- Every DFA has the following (required) properties:
  - states: a set of the DFA's valid states, each of which must be represented as a string
  - input_symbols: a set of the DFA's valid input symbols, each of which must also be represented as a string
  - transitions: a dict consisting of the transitions for each state. Each key is a state name and each value is a dict which maps a symbol (the key) to a state (the value).
  - initial_state: the name of the initial state for this DFA
  - final_states: a set of final states for this DFA
  - allow_partial: by default, each DFA state must have a transition to every input symbol; if allow_partial is True, you can disable this characteristic (such that any DFA state can have fewer transitions than input symbols)

Example

```
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'} )
```

```
if(dfa.accepts_input('10011')):
print("Accepted")
else:
print("Rejected")

Output
Accepted
```

class NFA(FA)

- The NFA class is a subclass of FA and represents a nondeterministic finite automaton.
- Every NFA has the same five DFA properties:
- state, input_symbols, transitions, initial_state, and final_states.

Example

```
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'}
)
```

```
if(dfa.accepts_input(‘abbba')):
print("Accepted")
else:
print("Rejected")
```

Output

Accepted

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')
```

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
```

Example 3

- **Regular expression of FA:** 101^+
- Accepted Inputs: 101, 1011, 10111, 10111
- Rejected Inputs: 100, 010, 000

```
def FA(s):
#if the length is less than 3 then it can't be accepted, Therefore end the process.
        if len(s)<3:
                return "Rejected"
#first three characters are fixed. Therefore checking them using index
        if s[0]=='1':
                if s[1]=='0':
                        if s[2]=='1':
# After index 2 only "1" can appear. Therefore break the process if any other character is detected
                                for i in range(3,len(s)):
                                        if s[i]!='1':
                                                return "Rejected"
                                return "Accepted"
                        return "Rejected"
                return "Rejected"
        return "Rejected"
inputs=['1','10101','101','10111','01010',""]
for i in inputs:
        print(FA(i))
```

Output

```
Rejected
Rejected
Accepted
Accepted
Rejected
Rejected
```

Example 4

- **Regular expression of FA**: (a+b)*bba
- Accepted Inputs: bba, ababbba, abba
- Rejected Inputs: abb, baba,bbb

```
def FA(s):
        size=0
#scan complete string and make sure that it contains only 'a' & 'b'
        for i in s:
                if i=='a' or i=='b':
                        size+=1
                else:
                        return "Rejected"
#After checking that it contains only 'a' & 'b'
#check it's length it should be 3 atleast
                if size>=3:
#check the last 3 elements
                        if s[size-3]=='b':
                                if s[size-2]=='b':
                                        if s[size-1]=='a':
                                                return "Accepted"
                                        return "Rejected"
                                return "Rejected"
                        return "Rejected"
        return "Rejected"
inputs=['bba', 'ababbba', 'abba', 'abb', 'baba', 'bbb','']
for i in inputs:
        print(FA(i))
```

Output

```
Accepted
Accepted
Accepted
Rejected
Rejected
Rejected
```

Example

- Design a deterministic finite automata (DFA) for accepting the language $L = \{a^n b^m | n \mod 2 = 0, m \geq 1\}$
- Regular expression for above language L is,

  $L = (aa)^*.b^+$

  Input: a a b b b

  Output: ACCEPTED // n = 2 (even) m=3 (>=1)

  Input: a a a b b b

  Output: NOT ACCEPTED // n = 3 (odd), m = 3

Example…

```
def start(c):
    if (c == 'a'):
        dfa = 1
    elif (c == 'b'):
        dfa = 3

    # -1 is used to check for any
    # invalid symbol
    else:
        dfa = -1
    return dfa
```

```
def state1(c):
    if (c == 'a'):
        dfa = 2
    elif (c == 'b'):
        dfa = 4
    else:
        dfa = -1
    return dfa
```

Example…

```
def state2(c):
    if (c == 'b'):
        dfa = 3
    elif (c == 'a'):
        dfa = 1
    else:
        dfa = -1
    return dfa
```

```
def state3(c):
    if (c == 'b'):
        dfa = 3
    elif (c == 'a'):
        dfa = 4
    else:
        dfa = -1
    return dfa
```

Example…

```
def state4(c):
    dfa = -1
    return dfa

def isAccepted(String):
        # store length of String
    l = len(String)
    dfa = 0
        for i in range(l):
```

```
 if (dfa == 0):
    dfa = start(String[i])

 elif (dfa == 1):
    dfa = state1(String[i])

 elif (dfa == 2) :
    dfa = state2(String[i])

 elif (dfa == 3) :
    dfa = state3(String[i])

 elif (dfa == 4) :
    dfa = state4(String[i])
 else:
    return 0
if(dfa == 3) :
return 1
else:
    return 0
```

Example…

```
# Driver code
if __name__ == "__main__" :
    String = "aaaaaabbbb"
    if (isAccepted(String)) :
        print("ACCEPTED")
    else:
        print("NOT ACCEPTED")
```

- **Output:**

  ACCEPTED

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 Σ = {0, 1}.

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

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 Σ = {0, 1}.

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