Advanced Programming

Finite Automata and Regular Expressions

Ramaseshan Ramachandran

1 Regular Expression

2 Finite Automata

OPERATIONS ON STRINGS

```
# Enclose letters, special characters, spaces,
# digits  using single quote or double quotation
# marks

first_name = "Arun"
last_name =  "Dravid"

# concatenate strings
greet = "hi " + ' ' + first_name + ' ' + last_name
print(greet)

#repetition of strings
repeat_3_times = greet*3
```

STRING AS ARRAY

```
my_string = "hello"

#length of a string
print(len(my_string))

#locate the character(s) using  indexes
print(my_string[1]) #output = 'e'

#slicing  - extracting parts of a sequence
print(my_string[2:4])   # output = 'll' ? What are the start
#  and end indexes used?
print(my_string[:4])    # output = 'hell'
print(my_string[2:])    # output = ?

#what does the following code do?
print(my_string([-1])
```

METHODS/FUNCTIONS TO OPERATE ON STRINGS

```
message = "HELLO WORLD"
lower_case = message.lower()
# Output: "hello world"
upper_case = lower_case.upper()
#output = 'HELLO WORLD'

# split() function is used to split two words uing a separator
words = "apple,banana"
fruit_list = words.split(",")
# Output: ["apple", "banana"]

# find whether a string exists in a sequence
# finds the first occurrence.
# If found, returns the position,
# else return -1

text = "Python is awesome"
position = text.find("awesome")
# Output: 12
```

STRINGS ...

```python
name = "Roy"
greeting = f"Hello, {name}!"
print(greeting)  # Output: Hello, Roy!
```

STRINGS ...

```
input_string = 'Hello, World'
upper_string = input_string.upper()
# Output: HELLO, WORLD!
lower_string = input_string.lower()
# Output: hello, world!
```

STRINGS ...

```python
input_string = 'Hello, World'
index = input_string.find("world")
# Output: 7 (index of the first occurrence)
```

STRINGS ...

```
# Checks if all characters are letters
is_alpha = input_string.isalpha()
#is_apha = True, if input_string contains
#alphabets [a-z]+, else False
# Checks if all characters are numbers
is_numeric = input_string.isnumeric()
```

STRINGS ...

```
text = 'Did you find Python in that corner?'
index = text.index("Python")  # Returns 13
#starts with a character or string
if text.startswith("Did"):
    print("String starts with 'Did'")
if 'that' in text:
    print("Substring 'that' found in the string")
#output: index of the last 'o'  = 29
does_text_ends_with = text.endswith('that corner?')
for in text:
    print(i)
    #prints all the characters in the text, including
    #the white space
```

REGULAR EXPRESSIONS

In a text environment, it is often required to search a string from a huge collection of documents. If we want a specific string, we use a simple search. There are situations not a specific string but a pattern requires to be found in the entire document. In such cases, *Regular Expressions* are useful.

- A language for specifying text search strings
- An algebraic notation for characterizing a set of strings a language for specifying text search strings
- A tool for pattern matching within text
- Precise description of a set of strings using a combination of alphabets/symbols
- Case sensitive search - String 'Hello' is different from 'hello'

EXAMPLES

- Search for dates in the corpus - not a specific date but a set of dates with varying values

  Find all strings that match "MM-DD-YYYY" 12-12-2024, 01-01-2022, 06-15-2021...
- Get all the currency values found in a document -$12389.23, $56789.12,...
- Find strings Hello and hello - [Hh]ello
- Find any digit in the text - It costs anywhere between **5** to **8** dollars

BASIC RULES OF REGEX

- [0123456789] specifies any one of digits - [] specifies **disjunction**
- [ABCDEF] specifies ____________?
- - (hyphen) specifies the range
- [0-9] indicates any digit from 0 to 9
- [A-Z] means ____________?
- [a-z] represents ____________?
- [^A-Z] - indicates negation/NOT - do **NOT** match an upper case letter

- ? can be used to match zero or one instance of the preceding character -colou?r will match both color and colour
- $*$ - zero or more instances of the preceding character or combinations - a$*$ - a, aa, aaa, aaaa, ... This is also known as **Kleene** $*$
- [ab]$*$ represents a set of strings $\{a, aaa, b, bbb, ab, ababab, abbbbb, \ldots\}$

BUILDING BLOCKS OF REGEX

Some more rules

- $[a\text{-}z][a\text{-}z]$ - represents ___________?
- $+$ - positive closure or Kleene $+$ - represents ONE or MORE
- $[a\text{-}z][0-9]+$
- . is a wildcard - $r.n$ matches run, ran, ron, ...
- $[']\{3\}\,.*\,[']\{3\}$ - matches any text between "' and "'
- **Alphabets** - Represent individual symbols like letters, numbers, and punctuation
- **Wild cards** - Match any single character (".") or any character set ("[]")
- **Quantifiers** - Specification for repetition of patterns - zero or more ("*"), one or more ("+"), or exactly n times ("n")
- **Anchors** - Match positions within a string like beginning ("^") or end ("$") -

ANCHORS

Regex	Match	Examples
^	The beginning of the sentence	^Once - Once upon a time
$	The end of a line	\w+$ - How are you?
\s	Any white space character	\s[a-z]\s I have a Python book
		\s[a-z]+\s I have a Python book
		\s[a-z]*\s I have a Python book
\S	Matches any visible character	
\b	word boundary	\b[a-z] - I have a Python book
\B	non word boundary	\B[a-z]\B - I have a Python book

REGEX EXAMPLES

Day (DD) Rules

1. Must be two digits
2. Must follow date semantics
3. It restricts the day part to valid values (01-30,31)
   - One of the many possible patterns $\hat{}(\underline{0[1\text{-}9]}|\underline{[12][0\text{-}9]}|\underline{3[01]})$

     0[1-9] - accepts 0 as the first digit and any digits from $0 \ldots 9$

     [12][0-9] - accepts either 1 or 2 as the first digitand any digit from $0 \ldots 9$

     3[01] - accepts 3 as the first and 0 or 1 as the second digit

Identifying numbers (scientific notation of digits, float, etc)
[+-]?[0-9]*[.]?[0-9]+([eE][+-]?[0-9]+)?

DAY REGEX - PYTHON CODE

```python
#include the module that contains all regex functions
import re
def check_day(input_pattern:str) -> bool:
    '''
    Check whether the given pattern matches the dates beteeen 01-31
    Anything not in this set should be considered as invalid
    '''
    date_pattern = re.compile(r"^(0[1-9]|[12][0-9]|3[01])$")
    # If date_pattern matches input_pattern, return True;
    # else return False
    if ( date_pattern.match(input_pattern)):
        return True
    else:
        return False

if __name__ == '__main__':
    print(check_day(input("Input date: ")))
    # Test cases
    for date in range(40):
        print(f'{date} - {check_day(str(date))}')
```

EMAIL PATTERN

```
def check_email_address(input_pattern:str) -> str:
    '''
    Check whether the given pattern matches
    the standard email address
    This may verify 75-85% of the email addresses
    '''
    email_pattern = re.compile(r"^[a-zA-Z0-9._%+-]+@[a-zA-Z0
   -9.-]+\.[a-zA-Z]{2,}$")
    if ( email_pattern.match(input_pattern)):
        return "a valid email address"
    else:
        return "an invalid email address"
```

APPLICATIONS

- **Text Search and Manipulation**: - All important applications where Find and replace specific patterns, extract data, validate formats are important
- **Compilers/Interpretors, Programs** - Define syntax, validate user input, perform data parsing
- **Web Technologies** - Build search engines, extract information from web pages, filter content

FINITE AUTOMATA - ALPHABETS AND STRINGS

Definition

The term *alphabet* denotes any set of symbols

Example

English alphabets and numbers
The set containing binary alphabet $\{0, 1\}$

Definition

The term *Language* denotes any set of strings formed using the alphabet

Example

$\{000, 111, 0101, 1110000, \ldots\}$
$\{a, b, aa, aaabbb, ababab, \ldots\}$

OPERATIONS ON LANGUAGES

We can apply concatenation, union, and closure (Kleene[1] and positive[2]) operations on *Languages*. Let L be a set of alphabets {A,B,...,Z, a,b,...,z} and D be set containing a set of 10 digts $\{0, 1, \ldots, 9\}$

Name	Operation	Example
Union	$L \cup D = \{s \mid s \in L$ or $\in D\}$	s,5,2
Concatenation	$LM = \{sd \mid s$ is in and $d \in D\}$	12s,e4
Kleene Closure	$L^* = \bigcup_{i=0}^{\infty} L^i$	ϵ, 0000,1111,00110011
Positive Closure	$L^+ = \bigcup_{i=1}^{\infty} L^i$	0,1,010101
ϵ operations	$\epsilon s = s,\ s\epsilon = s,\ s^* = (s \mid \epsilon)^*$	

[1] L^* denotes zero or more concatenation of L
[2] L^+ denotes one or more concatenation of L

Let us consider a language whose *identifiers* constructed using the following alphabet

Alphabet $\{A, B, \ldots, Z, a, b, \ldots, z, 0, 1, \ldots, 9\}$

The rules for constructing the identifier is given below

identifier $letter(letter \mid digit)^*.$

letter $a \mid b \mid c \mid \ldots z$

digit $0 \mid 1 \mid 2 \mid \ldots 9$

Explanation

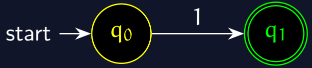
- $\mid$ is used in place of "*or*"
- The expression in $(letter|digit)^*$ represents zero or more of either the letter or digit

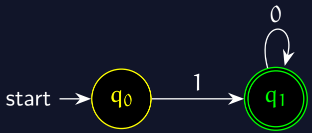
FINITE AUTOMATA

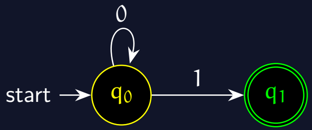
It is recognizer that determines whether a given string of alphabets adheres to the rules and patterns of a specific language. It functions as a gatekeeper, accepting only strings that conform to the language's grammar and structure, while rejecting those that do not. This is a mathematical model that consists of 5-tuple

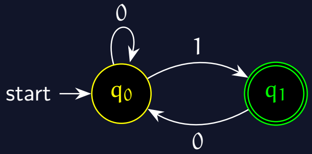
1. Q, a set of states
2. Σ, a set of input symbols/alphabets
3. q_0, a special start/initial state
4. F, a set of accepting states
5. $\delta : Q \times \Sigma$,a transition function that maps states-symbol pair to sets of states

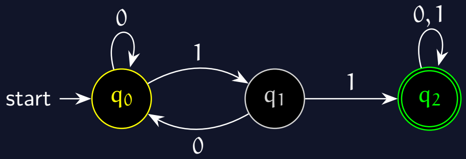
EXAMPLES OF FA

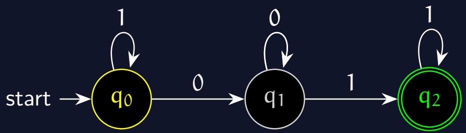


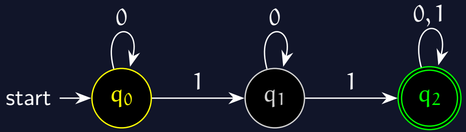






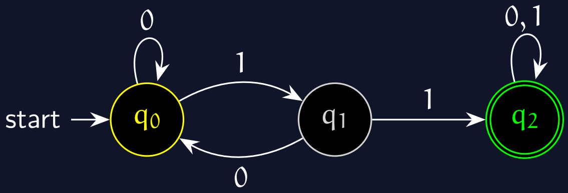






TRANSITION DIAGRAM

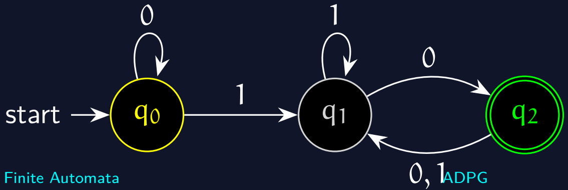
$$M = \{\{q_0, q_1, q_2\}, \{0, 1\}, \delta, q_0, \{q_2\}\}$$



	0	1
q_0	q_0	q_1
q_1	q_0	q_2
q_2	q_2	q_2

Transition Table

$$M = \{\{q_0, q_1, q_2\}, \{0, 1\}, \delta, q_0, \{q_1\}\}$$

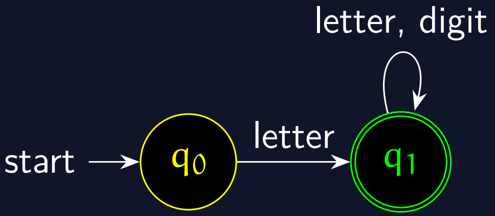


	0	1
q_0	q_0	q_1
q_1	q_0	q_2
q_2	q_1	q_1

Transition Table

Grammar

$$\begin{aligned} \text{identifier} &:: \text{letter}(\text{letter}|\text{digit})* \\ \text{letter} &:: [a\text{-}z] \\ \text{digit} &:: [0\text{-}9] \end{aligned}$$



	letter	digit
q_0	q_1	
q_1	q_1	q_1

DFA FOR VARIABLES IN PYTHON

$$\begin{aligned} \text{identifier} &:: (\_\text{letter})(\_|\text{letter}|\text{digit})* \\ \text{letter} &:: [a\text{-}z] \\ \text{digit} &:: [0\text{-}9] \end{aligned}$$

FA SIMULATION

```
function get_next_state(current_state, symbol)
{
    // read the transition table to get the next state corresponding
    // to the current state and the symbol
}

current_state := initial_state

for symbol in input_string:
    next_state = get_new_state((current_state, symbol))

    if next_state is None then
        return 'Invalid string for the automaton'

    current_state = next_state

    if current_state == accepting_state then
        return 'Valid string'
    else
        return 'Invalid string'
end
```

FA SIMULATION

```python
def is_string_accepted(transition_table:dict, initial_state:str,
   accepting_states, string) -> bool:
    current_state = initial_state

    for symbol in string:
        next_state = transition_table.get((current_state, symbol))
        if next_state is None:  # No transition for the given input
            return False

    current_state = next_state

    return current_state in accepting_states

if __name__ == '__main__':
    transition_table = {
        ('q0', '0'): 'q0', ('q0', '1'): 'q1',
        ('q1', '0'): 'q2', ('q1', '1'): 'q1',
        ('q2', '0'): 'q2', ('q2', '1'): 'q0'
    }

    initial_state = 'q0'
    # set of accepting states
    accepting_states = {'q2'}

    string = input('Input a binary string: ')

    if is_string_accepted(transition_table, initial_state, accepting_states,
   string):
        print('The string is accepted.')
    else:
        print('The string is not accepted')
```

EQUIVALENCE OF REGULAR EXPRESSION AND FINITE AUTOMATA

Definition (Deterministic Finite Automata - DFA)

Every step of a computation follows in a unique way from the preceding step

Definition (Nondeterministic Finite Automata - NFA)

- Several choices may exist for the next state at any point. Nondeterminism is a generalization of determinism
- Every nondeterministic finite automaton has an equivalent deterministic finite automaton
- A language is regular if and only if some regular expression describes it
- The Languages accepted by the finite automata are precisely the language denoted by the regular expressions.
- For every regular expression, there is precisely a NFA with ϵ-transition

FIND IDENTIFIERS IN A PYTHON PROGRAM

```
def find_identifier(line:str) ->list:
    identifier:list = []
    # Use regex rules to find the identifier(s)
    return identifier

if __name__ == '__main__':
    with open("tmp.py", "r") as f:
    # Read the entire text file
    # text = f.read()
    # Read the entire text file and
    # store all the lines into list
    # lines:list = f.readlines()

    #process the lines
    for line in f.readlines():
        find_identifier(line)
```