Regular expression

- regular expression or RegEx is a pattern matching/pattern specification language
- Used for searches, find, replace and validation
- A regular expression is a special sequence of characters that helps us match or find other strings or sets of strings, using a specialized syntax held in a pattern.
- regular expressions are declarative i.e. we tell the computer what and it figures out how.

Why Regular expression

- **They are fast: After some precalculation the re only has to look at each character in the input data once.**
- **It is readable: More readable than its procedural equivalent.**
- **Regular expressions are implemented using finite state machines.**

Useful applications of regular expressions

1. **Checking validity of formatted input:**
- Email,address,phone number,password
2. **Searching database**
3. **Searching text**
- for specific formatting
- Address, Date, Phone number, SSN
- for variantly spelled names
- for certain kinds of words
4. **Searching code**
- for uses of a certain function
- for uses of certain tags
5. **Parsing text for entry into a database**

- There are two types of characters in RegEx

1. Metacharacters: which have a special meaning when they are used in RegEx. e.g. square bracket, dot, ([], . , [^])
2. Regular characters: which have literal meaning

- Regular Expressions are used in programming languages to filter texts or text strings. It's possible to check, if a text or a string matches a regular expression.

Example	Description
[Pp]ython	Match "Python" or "python"
rub[ye]	Match "ruby" or "rube"
[aeiou]	Match any one lowercase vowel
[0-9]	Match any digit; same as [0123456789]
[a-z]	Match any lowercase ASCII letter
[A-Z]	Match any uppercase ASCII letter
[a-zA-Z0-9]	Match any of the above
[^aeiou]	Match anything other than a lowercase vowel
[^0-9]	Match anything other than a digit

Example	Description
duby?	Match “dub" or “duby": the y is optional
ruby*	Match "rub" plus 0 or more ys
ruby+	Match "rub" plus 1 or more ys
\d{3}	Match exactly 3 digits: \d\d\d
\d{3,}	Match 3 or more digits
\d{3,5}	Match 3, 4, or 5 digits

Example	Description
.	Match any character except newline
\d	Match a digit: [0-9]
\D	Match a nondigit: [^0-9]
\s	Match a whitespace character: [\t\r\n\f]
\S	Match nonwhitespace: [^ \t\r\n\f]
\w	Match a single word character: [A-Za-z0-9_]
\W	Match a nonword character: [^A-Za-z0-9_]

Pattern	description
\A or ^	Matches beginning of string.
\Z or $	Matches end of string. If a newline exists, it matches just before newline.
\z	Matches end of string.
\G	Matches point where last match finished.
\b	Matches word boundaries when outside brackets. Matches backspace (0x08) when inside brackets.
\B	Matches nonword boundaries.

Important Quote

A programmer is only as good as his knowledge of his language's libraries

- Python gives us two base methods to use our regular expression with. Syntax
- match(pattern, string, flags=0)
- checks to see if the pattern at the beginning of the string.
- returns a match object on success, None on failure.
- This function attempts to match RE pattern to string with optional flags.

re.match(pattern, string, flags=0)

- **pattern: This is the regular expression to be matched.**
- **string: This is the string, which would be searched to match the pattern at the beginning of string.**
- **flags: We can specify different flags using bitwise OR (|). These are called modifiers.**
- **Example: re.I represents ignore case matching**

- search(pattern, string, flags=0)
- search method checks for a match anywhere in the string and returns first matched string
- st="I am from kanpur from is"
- re.search("from",st) # match found
- re.findall(pattern, string)
- Returns a list of matched strings
- st="my lucky numbers are 1 and 9 and 42!!"
- re.findall("[0-9]+",st) #returns ['1', '9', '42‘]
- re.findall(r'pi?g', 'piiiigpgpig')) # ['pg', 'pig']

re.finditer()

- Return_an iterator yielding MatchObject instances over all non-overlapping matches for the RE pattern in string.
- The string is scanned left-to-right, and matches are returned in the order found.
- st = "123 456 ffgg 7890“
- for m in re.finditer("\d+", st):
- print(m.group(),"starts at index:", m.start())
- # 123 starts at index: 0
- # 456 starts at index: 4
- # 7890 starts at index: 13

Searching the specified string at the start and end of the given string

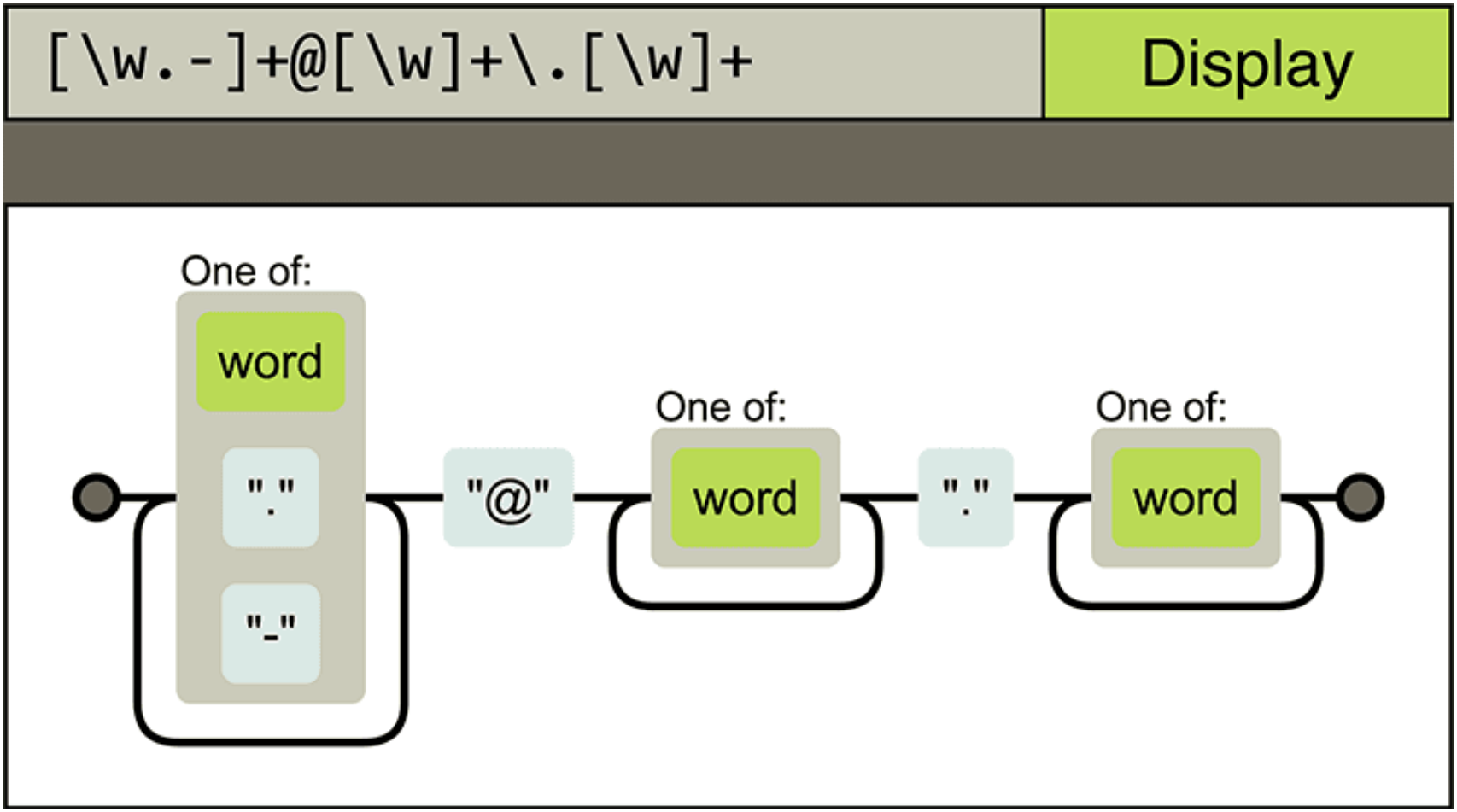
- \A or ^ used for searching the start of the specified string
- \Z or $ used for searching the end of the specified string
- st = """some thing is like
- someotherthing you know"""“
- print(re.search(“^some", st))
- print(re.search("know$", st))
- print(re.search(“\Asomeother", st,re.MULTILINE))
- print(re.search("know\Z", st))
- print(re.search("like$", st,re.MULTILINE))

Search at word boundary using \b and \B

- # searches word ‘the’
- m=re.search(r'the','bite on then dog')
- # word ‘the’ is not at the boundary so return None
- m=re.search(r'\bthe','bite onthe dog')
- # word ‘the’ is at the boundary so return None because \B searches only non boundary words
- m=re.search(r'\Bthe','bite then dog')
- # searches word ‘the’ because ‘the’ is at non-boundary
- m=re.search(r'\Bthe','bite another dog‘)

string = “computer science dean@psit.in, dept
dr.harsh_dev@gmail.com professor PSIT”

re.**findall**(r**'[\w.-]+@[\w]+\.[\w]+',** string)



program to extract names and the respective ages from large text

- str="""John is 15 years old, Mohan is currently 25 years old. Edvard is 57, and his grand father Michael is now 102 years old"""
- n=re.findall('[A-Z][a-z]+', str)
- a=re.findall('[0-9]+', str)
- #returns a list of pair of tuple of groups
- pair=re.findall('([A-Z][a-z]*)[a-z]*(\d{1,3})',str)
- Output: [('John', '15'), ('Mohan', '25'), ('Edvard‘, '57'),
- ('Michael', '102')]

match object returned by match or search

- match object instances also have several methods and attributes, the most important ones are:

Method	Purpose
group()	Return the string matched by the RE
groups()	Return a tuple containing all the subgroups
groupdict()	Return a dictionary containing all the named subgroups of the match, keyed by the subgroup name.
start()	Return the starting position of the match
end()	Return the ending position of the match
span()	Return a tuple containing the (start, end) positions of the match

- str = "banker 1\t2009-11-17\t1223.0\t2016-12-11"
- d1=re.search('(\d{4})-(\d{2})-(\d\d)',str)
- d1 is a match object, match object d1 instances also have several methods and attributes, the most important ones are:
- print('year=',d1.group(1),'\n','month=',d1.group(2),'\n‘
- ,'date=',d1.group(3), d1.groups())
- d1.start() #Return the starting position of the match
- d1.end() #Return the ending position of the match
- d1.span() #Return a tuple containing the (start, end) positions of the match

Concept of named group

- s = '''9-10-2017 and the 12345 11-12-2016 psit built in 8-12-2003'''

```
m=re.search('(?P<date>\d{1,2})-\
                    (?P<month>\d{2})-(?P<year>\d{4})',s)
```

- print(m.group('month'))
- print(m.group('year'))
- print(m.group('date'))
- print(m.groupdict())
- Output: {'date': '9', 'month': '10', 'year': '2017'}
- print('entire search in tuple',m.groups())
- Output: entire search in tuple ('9', '10', '2017')

- m = re.search(r"(?P<first_name>\w+) \
- (?P<last_name>\w+)", "Malcolm Reynolds")
- print(m.groupdict())

#{'first_name': 'Malcolm', 'last_name': 'Reynolds'}

(.+)/([A-Z][a-z]+) ([0-9]{1,2}),?([0-9]){4}/(.+)

- This pattern matches:

1. one or more chars
2. a slash
3. a single upper-case letter
4. one or more lower-case letter
5. a space
6. one or two digits
7. an optional comma
8. a space
9. exactly four digits
10. a slash
11. one or more char

(YEAR MONTH DATE)

- str=2010-01-01

  m=re.search(([0-9]{4})-([0-9]{2})-([0-9]{2}), str)
- print(m.group(1), m.group(2), m.group(3))
- str1=Jan 1, 2010

  m=re.search(([A-Z][a-z]+) ([0-9]{1,2}),? ([0-9]{4}), str1)
- print(m.group(3), m.group(1), m.group(2))

- \w word characters
- A word character is a character from a-z, A-Z, 0-9, including the _ (underscore) character.
- \s white-space characters
- White-Space Characters: Space, tab, linefeed, carriage-return, formfeed, vertical-tab, and newline characters are called "white-space characters" because they serve the same purpose as the spaces between words and lines on a printed page(they make reading easier).

Meaning of following control characters:

- 1. Carriage return
- 2. Line feed
- 3. Form feed
- Carriage return means to return to the beginning of the current line without advancing downward. The name comes from a printer's carriage, as monitors were rare when the name was coined. This is commonly escaped as "\r", abbreviated CR, and has ASCII value 13 or 0x0D.

- **Linefeed means to advance downward to the next line; however, it has been repurposed and renamed. Used as "newline", it *terminates* lines (commonly confused with *separating* lines). This is commonly escaped as "\n", has ASCII value 10 or 0x0A.**
- **Form feed means advance downward to the next "page". It was commonly used as page separators, but now is also used as section separators. (It's uncommonly used in source code to divide logically independent functions or groups of functions.) Text editors can use this character when you "insert a page break". This is commonly escaped as "\f", abbreviated FF, and has ASCII value 12 or 0x0C.**

re.sub(pattern, repl, string, count=0)

- **s=**"hello dgf hello fdh hello sjdsh"
- print(re.**sub**("hello","hi",**s**))
- **# will substitute first 2 occurences of hello by hi**
- print(re.**sub**("hello","hi",s,count=2))
- #Return a 2-tuple containing (new_string, **number**).
- # **number** is the number of substitutions that were made.
- print(re.**subn**("hello","hi",s))
- s="new delhi"
- print(re.**sub**("ne|de",'Hi',s))
- str = "yes I said yes I will Yes."
- print(re.**sub**("[yY]es","no", str))

changing date format

- st='05-14-2017'
- # changing date format mm-dd-yyyy -> dd-mm-yyyy
- print(re.sub('(\d{2})-(\d{2})-(\d{4})',r'\2-\1-\3',st))
- ----------
- # changing date format yyyyddmm -> dd/mm/yyyy
- s = '20121213'
- print(re.sub('(\d{4})(\d{2})(\d{2})', r'\2/\3/\1', s))

If repl is a callable, it's passed the Match object and must return a replacement string to be used

Changing mm-dd-yyyy To dd name of the month yyyy

```
def repl(m):
    months={'1':'Jan','2':'Feb','3':'March','4':'Apr','5':'May','
    6':'June‘,'7':'July','8':'Aug','9':'Sep','10':'Oct','11':'Nov','1
    2':'Dec'}
     return m.group(2)+' '+d[m.group(1)]+' '+m.group(3)
```

```
st =‘’’9-14-2017 and the 12345 11-12-2016 psit built in
•       8-12-2003’’’
print(re.sub(r'(\d{1,2})-(\d{2})-(\d{4})',repl, st))
# 14 Sep 2017 and the 12345 12 Nov 2016 psit built in 12 Aug 2003
```

Data wrangling or munging

- **Data wrangling** (sometimes referred to as **Data munging**) is the process of transforming and **mapping data from** one **"raw" data form into another format** with the intent of **making it more appropriate and valuable for a variety of downstream purposes such as analytics.** A **data wrangler** describes the person who performs these transformation operations.

Text Munging

- sub() **replaces** every occurrence of a pattern **with a string** or the **result of a function**.
- This example demonstrates using sub() with a function to “munge” text, or randomize the order of all the characters in each word of a sentence except for the first and last characters:

Word Scrambling program

```
import re, random
def repl(m):
    inner_word = list(m.group(2))
    random.shuffle(inner_word)
    return m.group(1) + "".join(inner_word) +
                                        m.group(3)
text = "Professor Abdolmalek, please report
your absences promptly."
print(re.sub(r"(\w)(\w+)(\w)", repl, text))
```

Greedy search

Laziness (Curb Greediness for Repetition Operators): *?, +?, ??, {m,n}?, {m,}?

- import re
- st="From colaaa: Using the: character"
- #Greedy matching tries to match as far as it can
- print(re.search("^F.+:",st))
- # ? can be used to stop the greedy search
- print(re.search("^F.+?:",st))

- s = "harsh you dev how are you"
- print(re.search("^h.*?you" ,s))

re.split()

- **split() splits a string into a list delimited by the passed pattern.**
- **The method is invaluable for converting textual data into data structures that can be easily read and modified by Python.**
- The following example creates a phonebook from a text which may come from a file.
- The entries are separated by one or more newlines.

- **text** = """Ross McFluff: 834.345.1254 155 Elm Street
  Ronald Heathmore: 892.345.3428 436 Finley Avenue

  Frank Burger: 925.541.7625 662 South Dogwood Way
  Heather Albrecht: 548.326.4584 919 Park Place"""“
- Now we convert the string into a list with each nonempty line having its own entry:
- Print(**re.split("\n+", text))**
- ['Ross McFluff: 834.345.1254 155 Elm Street', 'Ronald Heathmore: 892.345.3428 436 Finley Avenue', 'Frank Burger: 925.541.7625 662 South Dogwood Way', 'Heather Albrecht: 548.326.4584 919 Park Place‘]

- Finally, we split each entry into a list with **first name**, **last name**, **telephone number**, and **address**.
- We use the **maxsplit** parameter of **split()** because the address has spaces, our splitting pattern, in it:
- The **:?** pattern matches the colon after the last name, so that it does not occur in the result list.
- [re.split("**:?** ", entry, 3) **for** entry **in** entries]
- Output:
- **[['Ross', 'McFluff', '834.345.1254', '155 Elm Street'],** ['Ronald', 'Heathmore', '892.345.3428', '436 Finley Avenue'], ['Frank', 'Burger', '925.541.7625', '662 South Dogwood Way'], ['Heather', 'Albrecht', '548.326.4584', '919 Park Place']]

- line="surname: Obama, prename: Barack, profession: president"
- print(re.split(",* *\w*: ", line))
- **#['', 'Obama', 'Barack', 'president']**
- s="""Hello, This file: is going to get scrambled. I does'nrzt like Java. I like python. Java; is object oriented.""“”
- print(re.split(',|;|:|\.| ', s))
- print(re.split('[a-f]+',**'0aa34Bdcb9'**,flags=re.I))
- **#['0', '34', '9']**

re.escape(*pattern*)

- Escape all the characters in pattern except ASCII letters and numbers.
- This is useful if you want to match an arbitrary literal string that may have regular expression metacharacters in it. For example:
- **print(re.escape('python.exe?*'))**
- Output: python\.exe\?*
- legal_chars = string.digits + "!#$%&'*+-.^_`|~:"
- print('%s' % **re.escape**(legal_chars))
- # 0123456789\!\#\$\%\&\'*\+\-\.\^_\`\|\~\:

re.compile()

- ‘re’ module defines several functions, constants, and an exception. Some of the functions are simplified versions of the full featured methods for compiled regular expressions.
- Most non-trivial applications always use the compiled form.
- re.compile(*pattern*, *flags=0*)
- Compile a regular expression pattern into a regular expression object, which can be used for matching using its match() and search() methods.
- The expression’s behaviour can be modified by specifying a *flags* **value**.
- **Values** can be any of the following variables, combined using bitwise OR (the | operator).

- **ptrn = re.compile(pattern)**
- **result = ptrn.match(string)**
- **Above two statements are equivalent to**
- **result = re.match(pattern, string)**
- but using **re.compile()** and saving the resulting regular expression object for reuse **is more efficient when the expression will be used several times in a single program.**