Regular Expression - Introduction

- 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* provide an infrastructure for advanced text pattern matching, extraction and / or search – and – replace functionality.
- The Python module *re* (standard module for regexes) provides full support for regular expressions.
- The *re* module raises the exception *re.error* if an error occurs while compiling or using a regular expression.
- A regular expression pattern is a sequence of characters that will match sequences of characters in a target.
- When writing regular expression in Python, it is recommended that we use raw strings instead of regular Python strings. Raw strings with begin with a special prefix (r) and signal Python not to interpret backslashes and special metacharacters in the string, allowing us to pass them through directly to the regular expression engine.
- This means that a pattern like “\n\w” will not be interpreted and can be written as r”\n\w” instead of “\\n\\w” as in other languages.
- The patterns or regular expressions can be defined as follows:
  - Literal characters must match exactly. For example, “a” matches “a”.
  - Concatenated patterns match concatenated targets. For example, “ab” (“a” followed by “b”) matches “b”.
  - Alternate patterns (separated by a vertical bar) match either of the alternative patterns. For example, "(aaa)|(bbb)" will match either "aaa" or "bbb".
  - Repeating and optional items:
    - "abc*" matches "ab" followed by zero or more occurrences of "c", for example, "ab", "abc", "abcc", etc.
    - "abc+" matches "ab" followed by one or more occurrences of "c", for example, "abc", "abcc", etc, but not "ab".
    - "abc?" matches "ab" followed by zero or one occurrences of "c", for example, "ab" or "abc".

- Sets of characters: Characters and sequences of characters in square brackets form a set; a set matches any character in the set or range. For example, "[abc]" matches "a" or "b" or "c". And, for example, "[_a-z0-9]" matches an underscore or any lowercase letter or any digit.
- Groups: Parentheses indicate a group with a pattern. For example, "ab(cd)*ef" is a pattern that matches "ab" followed by any number of occurrences of "cd" followed by "ef", for example, "abef", "abcdef", "abcdcdef", etc.
- There are special names for some sets of characters, for example "\d" (any digit), "\w" (any alphanumeric character), "\W" (any non-alphanumeric character), etc.

Special Symbols and Characters [Also known as Metacharacters]

- 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.

Symbols	**Description**
Literal	Match literal string value *literal*
re1 | re2	Match regular expression *re1* or *re2*
.	Match *any character* except \n
^	Match *start of string*
$	Match *end of string*
*	Match *0 or more* occurrences of preceding regex
+	Match *1 or more* occurrences of preceding regex
?	Match *0 or 1* occurrences of preceding regex
{N}	Match *N* occurrences of preceding regex
{M, N}	Match *from M to N* occurrences of preceding regex
[…]	Match any single character from *character class.*
[..x-y..]	Match any single character in the *range from* x to y
[^…]	*Do not match* any character from character class, including any ranges, if present.
(*|+|?+{})?	*Apply"non-greedy"* versions of above occurrence / repetition symbols (*, +, ?, {})

Special Characters	**Description**
\d	Match any decimal *digit*, same as [0 – 9]
\D	*Do not match* any decimal digit. Inverse of \d
\w	Match any *alphanumeric* character, same as [A-Za-z0-9_].
\W	Inverse of \w
\s	Match *any whitespace character*. Same as [\n \t \r \v \f]
\S	Inverse of \s
\b	Match any *word boundary*.
\B	Inverse of \b
\N	Match saved *subgroup* N
\c	Match any *special character* c.
\A (\Z)	Match *start (end) of string*

- Matching More Than One Regex Pattern with Alternation (|):
    - Alternation is also sometimes known as *union or logical OR.* The pipe symbol (|), a vertical bar indicates an alternation operation. It is used to separate different regular expressions. For example, bat | bet | bit which matches string bat, bet, bit.
- Matching Any Single Character (.):
    - The dot symbol matches any single character except for \n. For example,
        - a.b which matches any character between “a” and “b” i.e. aab, abb, acb, a5b, a#b etc.
        - . . which matches any pair of characters.
        - . end which matches any character before string end.
        - Use backslash to specify dot character itself (i.e. “\.”).
- Matching from the Beginning or End of Strings (Word Boundaries) (^, $, \b, \B):
    - To match a pattern from beginning we must use the caret symbol (^) or \A and similarly, the dollar sign ($) or \Z to match a pattern from the end of a string. For Example,
        - ^Success will match any string that starts with Success.

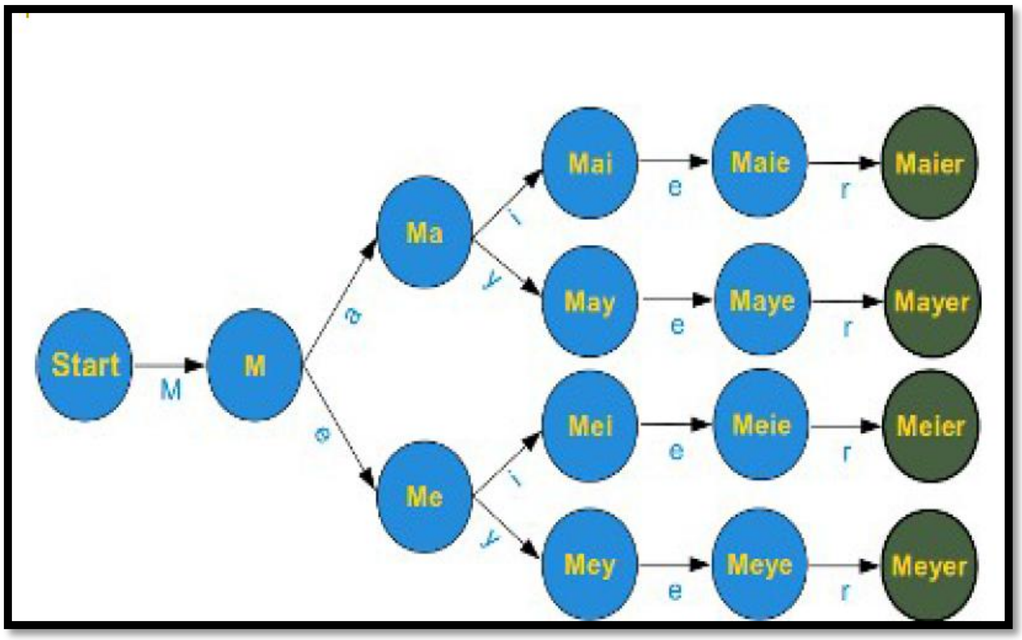
        - best$ will match any string that ends with best.
    - The \b and \B relate to word boundary matches. For Example,
        - \b the will match any string that starts with the
        - \b the \b will matches only the word the
        - \B the will matches any string that contains but does not begin with the.
- Creating Character Classes ([ ]):
    - The brackets symbols [ ] were invented where we want to match specific characters. For Example,
        - C[aeu]t will match for strings like Cat, Cet, Cut.
        - [cr] [23] [dp] [o2] will be string of four characters. First is c / r, second is 2 / 3, then d /p and then o / 2. It may be like c2do, r3p2, c3po etc.
- Denoting Ranges (-) and Negation (^):
    - A hyphen (-) between a pair of symbols enclosed in brackets is used to indicate a range of characters. For Example: A-Z, a-z, 0-9 etc.
    - If a caret (^) is the first character immediately inside the open left bracket, this symbolizes a directive not to match any of the characters in the given character set. For Example: [^aeiou] – Not to include *"aeiou"* (Allows only non – vowel characters or consonants).
- Multiple occurrence / repetition using closure operators (*, +,?, { }):
    - The special symbols *, + and ? can be used to match single, multiple or no occurrences of string patterns.
    - The asterisk (*) operator will match zero or more occurrences to its left. This operation is known as the *Kleene Closure*.
    - The plus (+) operator will match one or more occurrences of a regex. This operation is known as the *Positive Closure*.
    - The question mark (?) operator will match exactly 0 or 1 occurrences of a regex.
    - The brace operators ({ }) with a either a single value or a comma-separated pair of values indicate a match of exactly N occurrences (for {N}) or a range of occurrences i.e. M to N ({ M, N }).

- For Example:
  - [dn] ot? Will match “d” or “n” followed by an “o” and at most one “t” after that ; thus do, no, dot, not etc.
  - 0 ? [1 – 9] will match any numeric digit possibly prepended with a 0.
  - [0 – 9] {15, 16} will allow fifteen or sixteen digits (e.g. Credit Card Numbers)
  - </?[^ >] +> will match all valid / invalid HTML tags.

- Special Characters Representing Character Sets:
  - We can use special characters to represent character set. *d* is used to indicate the match of any decimal digit. *w* is used to denote the entire alphanumeric character class and \s is used for whitespace characters.
  - For Example:
    - \w+ - \d+ : alphanumeric string and number separated by a hyphen.
    - \d{3} - \d{3} - \d{4}: All digits with format 000-000-0000. [American format telephone number]
    - \w+@\w+\.com: Email Address format.
- Designating Groups with Parentheses (( )):
  - A pair of parentheses can accomplish either grouping regular expressions or matching subgroups. For Example,
    - \d + (\.\d*)?: represents simple string with floating point numbers.

- **Example:**
- Square brackets, “[“ and “]”, are used to include a character class.
- [xyz] means e.g. *either an “x”, an “y” or a “z”.*
- Consider following example:

  r"M[ae][iy]er”



Regexes and Python

- Python currently supports regular expressions through the *re* module, which was introduced and replaced the deprecated *regex* and *regsub* modules – both modules were removed from Python in version 2.5.
- The *re* module supports the more powerful and regular Perl-style regexes.
- The *re* Module: Core Functions and Methods
- Compiling Regexes with compile()
- Match Objects and the group() and groups() Methods
- Matching Strings with match()
- Looking for a Pattern within a String with search() (Search Vs. Match)
- Matching More than On String (|)
- Matching Any Single Character (.)
- Creating Character Classes ([ ])
- Repetition, Special Characters and Grouping
- Matching from the Beginning and End of Strings and on Word Boundaries
- Finding Every Occurrence with findall() and finditer()
- Searching and Replacing with sub() and subn()
- Splitting with split()
- Extension Notations (?...)

- <u>Match Method</u>
  - There are two types of Match Methods in RE
    - Greedy *(? ,*, +, {m,n})*
    - Lazy *(??, *?, +?, {m,n}?)*
  - Most flavors of regular expressions are greedy by default.
  - To make a Quantifier Lazy, append a question mark to it.
  - The basic difference is, Greedy mode *tries to find the last possible match*, lazy mode *the first possible match*.
  - Example of Greedy Match and Lazy Match:

```
#Greedy Match
import re
s = '<html><head><title>Title</title>'
len(s)
print(re.match('<.*>',s).span())

#Gives start and end position of the match.
print(re.match('<.*>',s).group())


#Lazy Match
print(re.match('<.*?>',s).group())
```

  - *Explanation:*
  - The RE matches the '<' in <html>, and the .* consumes the rest of the string. There's still more left in the RE, though, and the > can't match at the end of the string, so the regular expression engine has to backtrack character by character until it finds a match for the >. The final match extends from the '<' in <html>to the '>' in </title>, which isn't what you want.
  - In the Lazy (non-greedy) qualifiers *?, +?, ??, or {m,n}?, which match as little text as possible. In the example, the '>' is tried immediately after the first '<' matches, and when it fails, the engine advances a character at a time, retrying the '>' at every step.

- Match Function
    - The *match* function checks whether a regular expression matches are the beginning of a string. It is based on the following format:

      match (regular expression, string, [flag])
    - *Flag Values:*
        - re.IGNORECASE: There will be no difference between uppercase and lowercase letters.
        - Re.VERBOSE: Comments and spaces are ignored in the expression.
    - *Note that this method stops after the first match, so this is the best suited for testing a regular expression more than extracting data.*

Parameter	*Description*
pattern	Regular Expression to be matched.
string	String, which would be searched to match the pattern at the beginning of string.
flags	Can specify different flags using bitwise OR (|). These are modifiers. (Check Modifier Table)

    - The *re.match*() returns a *match* object on success, *None* on failure. We can use *group(num)* or *groups()* function of *match* object to get matched expression.

Match Object Methods	*Description*
group (num = 0)	Returns entire match or specific sub group num.
groups()	Returns all matching subgroups in a tuple.

- Search Function
    - This function searches for first occurrence of *RE pattern* within *string* with optional *flags*.

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

Parameter	*Description*
Pattern	Regular Expression to be matched.
String	is searched to match the pattern anywhere in the string.
Flags	Can specify different flags using bitwise OR (|).

- The *re.search*() returns a *match* object on success, *None* on failure. We can use *group(num)* or *groups()* function of *match* object to get matched expression.

Match Object Methods	*Description*
group (num = 0)	Returns entire match or specific sub group num.
groups()	Returns all matching subgroups in a tuple.

- Python offers two different primitive operations based on regular expressions: match checks for a match only at the beginning of the string, while search checks for a match anywhere in the string.

Modifier / Flags	
Modifier	Description
re.I re.IGNORECASE	Performs case-insensitive matching
re.L re.LOCALE	Interprets words according to the current locale. This interpretation affects the alphabetic group (\w and \W) and word boundary behavior (\b and \B).
re.M re.MULTILINE	Makes $ match the end of a line and makes ^ match the start of any line.
re.S re.DOTALL	Makes a dot match any character, including a newline.
re.u re.UNICODE	Interprets letters according to the Unicode character set. This flag affects the behavior of \w, \W, \b, \B.
re.x re.VERBOSE	It ignores whitespace.

- *Match Object Methods and attributes:*

Method	Purpose
group()	Returns the string matched by the RE
start()	Returns the starting position of the match.
end()	Returns the ending position of the match.
span()	Returns a tuple containing the (start, end) positions of the match.

```
#Example: Regular Expression - 1 (search())

import re
text = 'This is my First Regulr Expression Program'
patterns = [ 'first', 'that', 'program' ]
for pattern in patterns:
    print('Checking "%s" in "%s" ->' % (pattern, text))
    if re.search(pattern, text, re.I):
        print ('found a match!')
    else:
        print ('no match')
```

- Regular Expressions are also commonly used to modify strings in various ways with following pattern methods:

Method	Purpose
split()	Split the string into a list wherever the RE matches.
sub()	Find all substrings where the RE matches and replace them with a different string.
subn()	Does the same thing as sub(), but returns the new string and the number of replacements.
findall()	Finds all non-overlapping matches of *pattern* in *string.*
finditer()	Returns an iterator yielding MatchObject instances over all non-overlapping matches for the *pattern* in *string*.

- split Method: Regular expressions are compiled into pattern objects, which have methods for various operations such as searching for pattern matches or performing string substitutions.

  re.split (string, [maxsplit = 0])

- sub Method: This method replaces all occurrences of the *RE pattern* in *string* with *rep1,* substituting all occurrences unless *max* provided. This method will return modified string.

  re.sub (pattern, rep1, string, max=0)

- findall():
  - Returns all non-overlapping matches of *pattern* in *string,* as a list of strings.
  - The *string* is scanned left-to-right and matches are returned in the order found.
  - If one or more groups are present in the pattern, return a list of groups; this will be a list of tuples if the pattern has more than one group.
  - Empty matches are included in the result unless they touch the beginning of another match.

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

- 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.
  - Empty matches are included in the result unless they touch the beginning of another match.
  - Always returns a list.

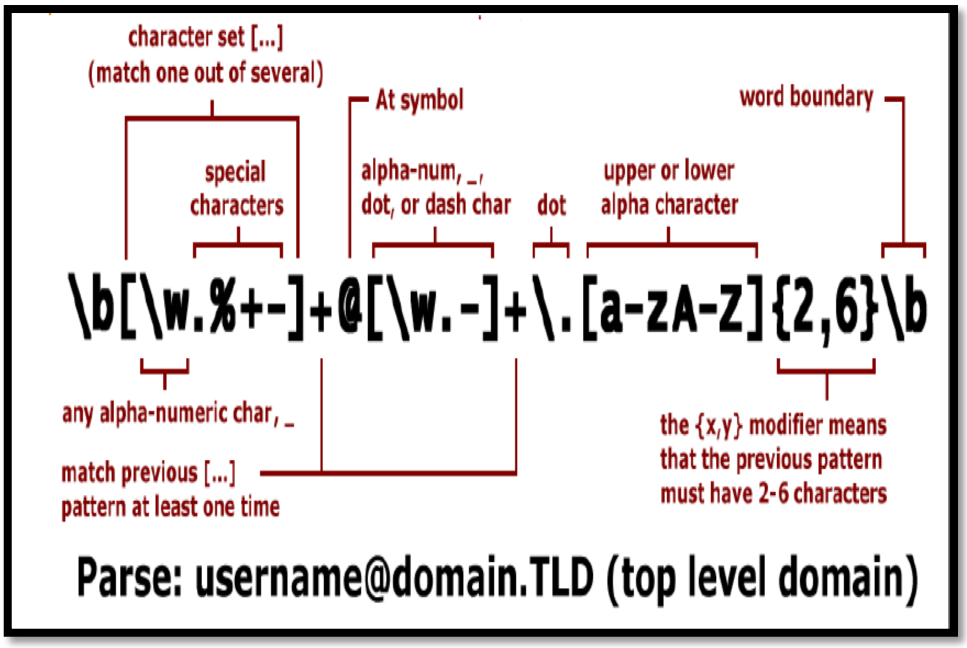
    re.finditer(patter, string, flags=0)

- re.compile():
  - If we want to use the same regexp more than once in a script, it might be a good idea to use a regular expression object, i.e. the regex is compiled.

    re.compile (pattern [, flags])

  - Compile returns a regex object, which can be used later for searching and replacing.

- The expressions behavior can be modified by specifying a flag value.
- Compile() compiles the regular expression into a regular expression object that can be used later with .search(), .findall() or .match().
- Compiles regular objects usually are not saving much time, because Python internally compiles AND CACHES regexes whenever we use them with re.search() or re.match().
- The only extra time a non-compiled regex takes is the times it needs to check the cache, which is a key lookup of a dictionary.


character set [...] (match one out of several)
special characters
At symbol
alpha-num, _, dot, or dash char
dot
upper or lower alpha character
word boundary
\b[\w.%+-]+@[\w.-]+\.[a-zA-Z]{2,6}\b
any alpha-numeric char, _
match previous [...] pattern at least one time
the {x,y} modifier means that the previous pattern must have 2-6 characters
Parse: username@domain.TLD (top level domain)

- Extension Notation:
  - Extension notations start with a (? but the ? does not mean 0 or 1 occurrence in this context.
  - Instead, it means that you are looking at an extension notation.
  - The parentheses usually represents grouping but if you see (? Then instead of thinking about grouping, you need to understand that you are looking at an extension notation.
  - <u>Unpacking an Extension Notation</u>
    - example: (?=.com)
    - step1: Notice (? as this indicates you have an extension notation
    - step2: The very next symbol tells you what type of extension notation.

- *Extension Notation Symbols:*
- (?aiLmsux):
  - (One or more letters from the set "i" (ignore case), "L", "m"(mulitline), "s"(. includes newline), "u", "x"(verbose).)
  - The group matches the empty string; the letters set the corresponding flags (re.I, re.L, re.M, re.S, re.U, re.X) for the entire regular expression.
  - This is useful if you wish to include the flags as part of the regular expression, instead of passing a flag argument to the compile() function.
  - Note that the (?x) flag changes how the expression is parsed.
  - It should be used first in the expression string, or after one or more whitespace characters. If there are non-whitespace characters before the flag, the results are undefined.
- (?...):
  - This is an extension notation (a "?" following a "(" is not meaningful otherwise).
  - The first character after the "?" determines what the meaning and further syntax of the construct is.
  - Extensions usually do not create a new group; (?P<name>...) is the only exception to this rule. Following are the currently supported extensions.
  - (?:…):
    - A non-grouping version of regular parentheses.
    - Matches whatever regular expression is inside the parentheses, but the substring matched by the group cannot be retrieved after performing a match or referenced later in the pattern.
  - (?P<name>…):
    - Similar to regular parentheses, but the substring matched by the group is accessible via the symbolic group name name.

- Group names must be valid Python identifiers, and each group name must be defined only once within a regular expression.
- A symbolic group is also a numbered group, just as if the group were not named. So the group named 'id' in the example above can also be referenced as the numbered group 1.
- For example, if the pattern is (?P<id>[a-zA-Z_]\w*), the group can be referenced by its name in arguments to methods of match objects, such as m.group('id') or m.end('id'), and also by name in pattern text (for example, (? P=id)) and replacement text (such as \g<id>).

▪ (?P=name):

- Matches whatever text was matched by the earlier group named name.

▪ (?#...):

- A comment; the contents of the parentheses are simply ignored.

▪ (?=...):

- Matches if ... matches next, but doesn't consume any of the string. This is called a lookahead assertion.
- For example, Isaac (?=Asimov) will match 'Isaac ' only if it's followed by 'Asimov'.

▪ (?!...):

- Matches if ... doesn't match next. This is a negative lookahead assertion.
- For example, Isaac (?!Asimov) will match 'Isaac ' only if it's not followed by 'Asimov'.

▪ (?<=...):

- Matches if the current position in the string is preceded by a match for ... that ends at the current position.

        - This is called a positive lookbehind assertion. (?<=abc)def will match "abcdef", since the lookbehind will back up 3 characters and check if the contained pattern matches.
        - The contained pattern must only match strings of some fixed length, meaning that abc or a|b are allowed, but a* isn't.
    - (?<!...):
        - Matches if the current position in the string is not preceded by a match for ....
        - This is called a negative lookbehind assertion. Similar to positive lookbehind assertions, the contained pattern must only match strings of some fixed length.

- re.purge():
    - Clear the regular expression cache.
    - The re module keeps a cache of implicitly compiled regular expression patterns. The number of patterns cached differs between Python versions, with more recent versions generally keeping 100 items in the cache.
    - When the cache space becomes full, it is flushed automatically.

A Longer Regex Example (like Data Generators, Matching a string etc.):

- We will now run through an in-depth example of the different ways to use regular expressions for string manipulation.
- The first step is to come up with some code that actually generates random data on which to operate.
- We need a script that generates a data set - the program simply displays the generated set of strings to standard output, this output could vary. That is, the script creates random data for regular expressions practice.
- The following script generates strings with three fields, delimited by a pair of colons or double colons.
    - The first field is a random (32-bit) integer, which is converted to a date.
    - The next field is a randomly generated e-mail address.
    - The last field is a set of integers separated by a single dash / hyphen (-).

Data Generators Example

```
from random import randrange, choice
from string import ascii_lowercase as lc
from sys import maxsize
from time import ctime
tlds = ('com', 'edu', 'net', 'org', 'gov')
for i in range(randrange(5, 11)):
    dtint = randrange(maxsize) # pick date
    dtstr = ctime(dtint) # date string
    llen = randrange(4, 8) # login is shorter
    login = ''.join(choice(lc) for j in range(llen))
    dlen = randrange(llen, 13) # domain is longer
    dom = ''.join(choice(lc) for j in range(dlen))
    print('%s::%s@%s.%s::%d-%d-%d'%(dtstr,login,dom,
          choice(tlds), dtint, llen, dlen))
```

- Explanation of code:
    - For above example, we require the use of multiple modules. So, we choose to import only specific attributes from these modules.
    - *tlds* is a simply a set of higher-level domain names from which we will randomly pick for each randomly generated e-mail address.
    - Each time our script executes, between 5 and 10 lines of output are generated (as we have generates random numbers using random.randrange() from 5 to 11).
    - For each line, we choose a random integer from the entire possible range (0 to $2^{31} – 1$ [sys.maxsize]), and then convert that integer to a date by using time.ctime(). System time in Python and most POSIX-based computers is based on the number of seconds that have elapsed since the “epoch,” which is midnight UTC/GMT on January 1, 1970. If we choose a 32-bit integer, that represents one moment in time from the epoch to the maximum possible time, 2^{32} seconds after the epoch.

- The login name for the fake e-mail address should be between 4 and 7 characters in length (thus randrange(4, 8)). To put it together, we randomly choose between 4 and 7 random lowercase letters, concatenating each letter to our string, one at a time. The functionality of the random.choice() function is to accept a sequence, and then return a random element of that sequence. In our case, the sequence is the set of all 26 lowercase letters of the alphabet, string.ascii_lowercase.
- We decided that the main domain name for the fake e-mail address should be no more than 12 characters in length, but at least as long as the login name. Again, we use random lowercase letters to put this name together, letter by letter.
- The key component of our script puts together all of the random data into the output line. The date string comes first, followed by the delimiter. We then put together the random e-mail address by concatenating the login name, the "@" symbol, the domain name, and a randomly chosen high level domain. After the final double-colon, we put together a random integer string using the original time chosen (for the date string), followed by the lengths of the login and domain names, all separated by a single hyphen.

- Matching a String:
    - To test the regex before putting it into an application, we will import the *re* module and assign one simple line from the output to a string variable 'data'.

```
>>> import re
>>> data = 'Thu Jan 11 08:41:21 2024::bvue@edxr.net::1704942681-4-4'
```

    - *<u>In our first example</u>*, we will create a regular expression to extract only the days of the week from the timestamps from each line of the data. We will use the following regex:

```
"^Mon|^Tue|^Wed|^Thu|^Fri|^Sat|^Sun"
```

    - This requires all day string with caret operators. We can bypass all the caret operators with a single caret if we group the day strings like this:

```
"^(Mon|Tue|Wed|Thu|Fri|Sat|Sun)"
```

- The parentheses around the set of strings mean that one of these strings must be encountered for a match to succeed. Due to parentheses we can take advantage of the fact that we can access the matched string as a subgroup:

```
>>> import re
>>> data = 'Thu Jan 11 08:41:21
            2024::bvue@edxr.net::1704942681-4-4'
>>> patt = '^(Mon|Tue|Wed|Thu|Fri|Sat|Sun)'
>>> m = re.match(patt, data)
>>> m.group()   #entire match
'Thu'
>>> m.group(1)  #subgroup 1
>>> m.groups()   # all subgroups
('Thu',)
```

- Both of the above regex are the most restrictive, specifically requiring a set number of strings. This might not work well in an internationalization environment, where localized days and abbreviations are used.
- A looser regex would be: ^\w{3}. This requires only that a string begin with three consecutive alphanumeric characters. And to translate the regex into English, the caret indicates 'begins with', the \w means any single alphanumeric character, and the {3} means that there should be 3 consecutive copies of the regex. Finally if we want grouping, parentheses should be used, such as ^(\w{3}):

```
>>> patt = '^(\w{3})'
>>> m = re.match (patt, data)
>>> if m is not None: m.group()

'Thu'
>>> m.group(1)
'Thu'
```

- A regex of ^(\w){3} is not correct as the {3} was inside the parentheses, the match for three consecutive alphanumeric characters was made first, and then represented as a group. Moving {3} outside, it is now equivalent to three consecutive single alphanumeric characters:

```
>>> patt = '^(\w){3}'
>>> m = re.match (patt, data)
>>> if m is not None: m.group()

'Thu'
>>> m.group(1)
'u'
```

- The reason why only the "u" shows up when accessing subgroup 1 is that subgroup 1 was being continually replaced by the next character. In other words, m.group(1) started out as "T," then changed to "h," and then finally was replaced by "u." These are three individual (and overlapping) groups of a single alphanumeric character, as opposed to a single group consisting of three consecutive alphanumeric characters.
- *In our next example*, we will create a regular expression to extract the numeric fields found at the end of each line of output. That is, we are looking for a set of three integers delimited by hyphens. We have a choice of using either search or match.
    - Initiating a search makes more sense because we know exactly what we are looking for (a set of three integers), that what we seek is not at the beginning of the string, and that it does not make up the entire string.

```
➔ Testing regex by using search():
>>> import re
>>> data = 'Thu Jun  5 07:08:47
1980::chrjw@mgwtvlxmvs.org::329017127-5-10'
>>> patt = '\d+-\d+-\d+'
>>> re.search(patt, data).group()
'329017127-5-10'
```

- If we were to perform a match, we would have to create a regex to match the entire line and use subgroups to save the data we are interested in. Following example uses match() with .+ to indicate just an arbitrary set of characters.

```
➔ Testing regex by using match():
>>> import re
>>> data = 'Thu Jun  5 07:08:47
1980::chrjw@mgwtvlxmvs.org::329017127-5-10'
>>> patt = '.+\d+-\d+-\d+'
>>> re.match(patt, data).group()
'Thu Jun  5 07:08:47
1980::chrjw@mgwtvlxmvs.org::329017127-5-10'
```

The above example works perfect, but we want the number fields at the end, not the entire string. So, we have to use parentheses to group we want:

```
>>> import re
>>> data = 'Thu Jun  5 07:08:47
1980::chrjw@mgwtvlxmvs.org::329017127-5-10'
>>> patt = '.+(\d+-\d+-\d+)'
>>> re.match(patt, data).group(1)
'7-5-10'
```

What happened? We should have extracted `329017127-5-10`, not just `7-5-10`. Where is the rest of the first integer? The problem is that regular expressions are inherently greedy. This means that with wildcard patterns, regular expressions are evaluated in left-to-right order and try to "grab" as many characters as possible that match the pattern. In the preceding case, the .+ grabbed every single character from the beginning of the string, including most of the first integer field that we wanted. The \d+ needed only a single digit, so it got "7," whereas the .+ matched everything from the beginning of the string up to that first digit:

“Thu Jun 5 07:08:47 1980::chrjw@mgwtvlxmvs.org ::329017127-5-10”.

One solution is to use the “don’t be greedy” operator ? after +. It directs the regular expression engine to match as few characters as possible. Solving greedy problem with non – greedy operator:

```
  >>> import re
  >>> data = 'Thu Jun  5 07:08:47
  1980::chrjw@mgwtvlxmvs.org::329017127-5-10'
  >>> patt = ‘.+?(\d+-\d+-\d+)’
  >>> re.match(patt, data).group(1)
'329017127-5-10'
```

Another solution is to use “::” field separator instead of “+”.

One final example, suppose that we want to pull out only the middle integer of the three-integer field.

```
>>> import re
>>> data = 'Thu Jun  5 07:08:47
1980::chrjw@mgwtvlxmvs.org::329017127-5-10'
>>> patt = ‘-(\d+)-’
>>> re.search(patt, data)
‘-5-‘
>>> m.group(1)
‘5’
```

Text Processing: Comma Separated Values, JSON, Python and XML

- Regardless of what type of applications we create, certainly, we will need to process human-readable data, which is referred to generally as *text*.
- Python’s standard library provides three text processing modules and packages to help us: *csv, json and xml*.
- *Comma Separated Values (csv):*
    - o Using CSVs is a common way to move data into and out of spreadsheet applications in plain text.

- The CSV module is useful for working with *data exported from spreadsheets and databases into text files formatted with fields and records,* commonly referred to as comma-separated value (CSV) format because commas are often used to separate the fields in a record.
- The CSV format is a very common data format used by various programs and applications as a format to export or import data.
- Most spreadsheet and database applications offer export and import facilities for CSV files.
- CSV files are simple, lacking many of the features of an Excel spreadsheet. For example, CSV files:
    - Don’t have types for their values – everything is a string.
    - Don’t have settings for font size or color.
    - Don’t have multiple worksheets.
    - Can’t specify cell widths and heights
    - Can’t have merged cells.
    - Can’t have images or charts embedded in them.
- The advantage of CSV files is simplicity.
- In short, CSV is just a text file of comma-separated values .
- The contents of CSV files are just rows of string values delimited by commas. Python has CSV parsing and generating library (module) *(import csv)* which can be used to writing data to CSV file and reading the data from CSV file.

```
#CSV Writing and Reading Example - 1
import csv
DATA=(
    (9,'Web Clients','base64,urllib'),
    (10,'Web Programming','cgi, time'),
    (11,'Text Processing','csv, json, xml'),
)
f = open('names.csv','w')
writer = csv.writer(f)
for record in DATA:
    writer.writerow(record)
f.close()

f=open('names.csv','r')
reader = csv.reader(f)
```

```
print('Chapter',''.ljust(15),'Name',''.ljust(32),
      'Featuring ',''.ljust(4))
print('==============================================')
for r in reader:
    if len(r) == 3:
        line = '%s %s %s' % (r[0].ljust(20),
               r[1].ljust(40),r[2].ljust(4))
        print (line)
f.close()
```

- In above example, we first import the *csv* module. Following the import statements is our data set. This is made up of 3 – tuples that have columns like chapter numbers, name, featuring.
- Six lines after data set is to write data in csv file. *csv.writer()* is a function that takes an open file object and returns a writer object. The writer features a writerow() method, which we use to output lines or rows of comma-separated data to the open file. After this the file is closed.
- After writing sections, reading section code is started. *csv.reader()* is the opposing function which returns an iterable object that we can use to read in and parse each row of CSV data. Like *csv.writer()*, *csv.reader()* also takes an open file handle and returns a reader object. When we iterate through each row of data, the CSVs are automatically parsed and returned to us. We display the output then close the file when all rows have been processed.

- In addition to *csv.reader()* and *csv.writer(),* the *csv* module also features the *csv.DictReader* and *csv.DictWriter* classes which read CSV data into a dictionary and write dictionary fields to a CSV file.

```
#CSV Writing and Reading Example - 2
import csv

f = open('bookdata.csv','w')
fn=["Name", "Address", "Phone"]
writer = csv.DictWriter(f,fieldnames = fn)
writer.writeheader()
writer.writerow({'Name':'Mike','Address':'USA',
                 'Phone':'12345'})
```

```
writer.writerow({'Name':'Frank','Address':'NJ',
                  'Phone':'98765'})

print('Data Written to CSV successfully')
print('--------------------------------')
f.close()

f = open('bookdata.csv','r')
r = csv.DictReader(f)
print ('==== ======== =========')
print ('Name Address  Phone')
print ('==== =======  =========')
for row in r:
    print(row['Name'], '', row['Address'],
          '   ', row['Phone'])
f.close()
```

- *JavaScript Object Notation (JSON):*
    - JSON is a lightweight text-based open standard designed for human readable data interchange alternative to XML and based on the ECMA – 262 standards.
    - JSON comes from the world of JavaScript – it's a subset of the language used specifically to pass around structured data.
    - Conventions used by JSON are known to programmers which include C, C++, Java, Python, Perl etc.
    - The format of JSON was specified by *Douglas Crockford.*
    - *Characteristics of JSON:*
        - JSON is easy to read and write.
        - It is a lightweight text – based interchange format.
        - JSON is language independent.
    - JSON is not a *document* format or a markup language or a programming language.
    - Support for JSON was officially added to the standard library in Python 2.6 via the *JSON* module.
    - The *JSON* module includes encoder and decoder classes, from which we can derive or use directly.

- To encode Python objects as JSON strings and decode JSON strings into Python objects, JSON module has dump() and load() functions in python.
- JSON Object has Data Types: Object (An Unordered key / value pairs wrapped in { }) and Array (An Ordered key / value pairs wrapped in [ ]).
- JSON Object Syntax has
  - Unordered sets of name / value pairs
  - Begins with { and ends with }.
  - Each name is followed by colon (:).
  - Name / value pairs are separated by comma (,).
  - Arrays in JSON are an ordered collection of values which begins with [and ends with].
- JSON Example:

```
employeeData = {
        “employe_id” : 1234567,
        “name” : “Mike Sherill”,
        “hire_date” : “01/01/2019”
        “location” : “Baroda”
        “random_nums” : [24, 65, 12, 94] //Arrays in JSON.
        };
```

- A JSON object is extremely similar to Python dictionary. Check following code snippets, in which we use a *dict* to transfer data to JSON object and then back again:

```
>>> dict(zip(‘abcde’, range(5)))
{'a': 0, 'b': 1, 'c': 2, 'd': 3, 'e': 4}
>>> json.dumps(dict(zip(‘abcde’, range(5))))
'{"a": 0, "b": 1, "c": 2, "d": 3, "e": 4}'
json.loads(json.dumps(dict(zip(‘abcde’,range(5)))))
{'a': 0, 'b': 1, 'c': 2, 'd': 3, 'e': 4}
```

- Python *dicts* are converted to JSON objects, similarly, Python *lists* / *tuples* are considered JSON arrays:

```
>>> list('abcde')
['a', 'b', 'c', 'd', 'e']
>>> json.dumps(list('abcde'))
'["a", "b", "c", "d", "e"]'
>>> json.loads(json.dumps(list('abcde')))
['a', 'b', 'c', 'd', 'e']
```

Key differences between JSON and Python Types		
JSON	**Python 2**	**Python 3**
object	dict	dict
array	list, tuple	list, tuple
string	unicode	str
number (int)	int, long	int
number (real)	float	float
true	True	True
false	False	False
null	None	None

- Another difference is that JSON does not use single quotes / apostrophes; every string is delimited by using double quotes. Also there are not extra trailing commas that Python Programmers casually place at the end of each sequence of mapping element for convenience.

```
from json import dumps
from json import loads
from pprint import pprint
emp = {
 '1': {
 'name': 'Mac',
 'salary': 30000,
 'join_year': 2007,
 },
 '2': {
 'name': 'Jack',
 'education': ['NET', 'SLET', 'Ph.D.'],
 'year': 2009,
 },
 '3': {
 'name': 'Hac',
 'year': 2009,
 }, }
```

```
print('*** RAW DICT ***')
print(emp)

print('\n*** PRETTY_PRINTED DICT ***')
pprint(emp)

print('\n*** RAW JSON ***')
print(dumps(emp))

print('\n*** PRETTY_PRINTED JSON ***')
print(dumps(emp, indent=5))

print('\n*** DECODED JSON ***')
print(loads(dumps(emp)))
```

- *Python and XML:*
    - XML stands for Extensible Markup Language – is a markup language much like HTML or SGML.
    - XML is recommended by the WWW Consortium and available as an open standard.
    - XML is extremely useful for keeping track of small to medium amounts of data without requiring a SQL – based backbone.
    - The first XML specification was published in 1998; the most recent update was released in 2008.
    - XML is a subset of SGML.
    - Python's original support for XML occurred with the release of version 1.5 and the *xmllib* module. Since then, it have evolved into the *xml* package, which provides a variety of ways to both parse as well as construct XML documents.
    - Python supports both document object model (DOM) tree – structured as well as event – based Simple API for XML (SAX) processing of XML documents.
    - The current version of the SAX specification is 2.0.1, so Python's support generally refers to this as SAX2.
    - SAX is a streaming interface, meaning that the documents are parsed and processed one line at a time. We can neither backtrack nor perform random access within an XML Document.

```
#Converting Python Dict to XML
from    xml.etree.ElementTree    import    Element,
SubElement, tostring
from xml.dom.minidom import parseString
emp = {
 '1': {
 'name': 'Mac',
 'salary': 30000, 'year': 2007,
 },
 '2': {
 'name': 'Jack',
 'education':  ['NET',  'SLET',  'Ph.D.'],  'year':
2009,
 },
 '3': {
 'name': 'Hac',  'year': 2009,
 },
}


e1 = Element('EMP_LIST')
for isbn, info in emp.items():
    e_list = SubElement(e1, 'emp')
    info.setdefault('salary', '15000')
    info.setdefault('education', 'BCA')
    for key, val in info.items():
        SubElement(e_list, key).text = ',   '
                 .join(str(val).split(':'))

xml = tostring(e1,encoding="unicode")
print ('*** RAW XML ***')
print (xml)

print ('\n*** PRETTY-PRINTED XML ***')
dom = parseString(xml)
print (dom.toprettyxml(' '))

print ('*** FLAT STRUCTURE ***')
for elmt in e1.getiterator():
     print (elmt.tag, '-', elmt.text)

print ('\n*** TITLES ONLY ***')
for e_lst in e1.findall('.//name'):
    print (e_lst.text)
```