Notation	Description	Example Regex
Symbols		
Literal	Match literal string value *literal*	foo
re1|re2	Match regular expressions *re1 or re2*	foo|bar
.	Match *any character* (except \n)	b.b
^	Match *start of string*	^dear
$	Match *end of string*	/bin/*sh$
*	Match *0 or more* occurrences of preceding regex	[a-za-z0-9]*
+	Match *1 or more* occurrences of preceding regex	[a-z]+\.com
?	Match *0 or 1* occurrences of preceding regex	goo?
{N}	Match *N* occurrences of preceding regex	[0-9]{3}
{M,N}	Match from *M to N* occurrences of preceding regex	[0-9]{5,9}
[...]	Match any single character from *character class*	[aeiou]
[..x-y..]	Match any single character in the *range from x to y*	[0-9],[A-Za-z]
[^...]	*Do not match* any character from character class, including any ranges, if present	[^aeiou] [^A-Za-z0-9_]
(*|+|?|{})?	Apply "non-greedy" versions of above occurrence/ repetition symbols (*, +, ?, {})	.*?[a-z]
(...)	Match enclosed regex and save as *subgroup*	([0-9{3}])?, f(oo|u)bar
Special Characters		
\d	Match any decimal *digit*, same as [0-9] (\D is inverse of \d: do not match any numeric digit)	data\d+.txt
\w	Match any alphanumeric character, same as [A-Za-z0-9_] (\W is inverse of \w)	[A-Za-z_]\w+
\s	Match any whitespace character, same as [\n\t\r\v\f] (\S)	of\sthe
\b	Match any *word boundary* (\B is inverse of \b)	\bThe\b
\N	Match saved *subgroup N* (see (...) above)	Price: \16
\c	Match any *special character* c verbatim (i.e., without its special meaning, literal)	\., \\, *
\A (\Z)	Match *start (end) of string* (also see ^ and $ above)	\ADear
Extension Notation		
(?iLnsux)	Embed one or more special "flags" parameters within the regex itself (vs. via function/method)	(?x), (?im)
(?:...)	Signifies a group whose match is *not* saved	(?:\w\.)*
(?P<name>...)	Like a regular group match only identified with name rather than a numeric ID	(?P<data>)
(?P=name)	Matches text previously grouped by (?P<name>) in the same string	(?P<data>)
(?#...)	Specifies a comment, all contents within ignored	(?#comment)
(?=...)	Matches if ... comes next without consuming input string; called *positive lookahead assertion*	(?=.comt)
(?!...)	Matches if ... does not come next without consuming input; called *negative lookahead assertion*	(?!.net)
(?<=...)	Matches if ... comes prior without consuming input string; called *positive lookbehind assertion*	(?<=800-)
(?<!...)	Matches if ... does not come prior without consuming input; called *negative lookbehind assertion*	(?<!192\.168\.)
(?(id/name)Y|N)	Conditional match of regex Y if group with given id or name exists else N; |N is optional	(?(1)y|x

Exercise

Regular Expressions. Create Regular expressions in Exercises 1-1 to 1-12 that:

1-1 Recognize the following strings: "bat", "bit", "but", "hat", "hit", or "hut"
Solution:
[bh][aiu]t

1-2. Match any pair of words separated by a single space, that is, first and last.
Solution:
[A-Za-z]+\s[A-Za-z]+

1-3. Match any word and single letter separated by a comma and single space, as in last name, first initial.
Solution:
[A-Za-z]+,\s[A-Za-z]+

1-4. Match the set of all valid Python identifiers.
Solution:
**[A-Za-z]\w*

1-5. Match a street address according to your local format (such as 1180 Bordeaux Drive)
Solution:
\d{4}\s[A-Z][a-z]+\s[A-Z][a-z]+

1-6. Match simple Web domain names that begin with "www." and end with a ".com" suffix; for example, www.yahoo.com. (other high-level domain names)
Solution:
r'www.(\w+)(.com|.net|.edu)'

1-7. Match the set of string representations of all Python integers.
Solutions
(-)?(\d)+

1-8. Match the set of the string representations of all Python longs.
Solutions:
-?\d+[Ll]

1-9. Match the set of string representations of all Python floats.
Solution:
r'\d+.\d*'|r'\d*.\d+'

1-10. Match the set of string representations of all Python complex numbers.
Solution:
'\d+.\d+[jJ]'

1-11. Match the set of all valid e-mail addresses (start with a loose regex, and then try to tighten it as much as you can, yet maintain correct functionality)

Solution:

r'(\w+)@(\w+).com'

1-12. Match the set of all valid Web site address (URLs) (start with a loose regex, and then try to tighten it as much as you can, yet maintain correct functionality).

Solution:

'(http|https|ftp|mailto|ldap|file|news|gopher|telnet)://(w+/)+w+'

1-13. type(). The type() built-in function returns a type object, which is displayed as the following Pythonic-looking string:

```
>>> type(0)
<type 'int'>
>>> type(.34)
<type 'float'>
>>> type(dir)
<type 'builtin_function_or_method'>
>>>
```

Create a regex that would extract the actual type name from the string. Your function should take a string like this <type 'int'> and return it. (Ditto for all other types, such as 'float', 'builtin_function_or_method', etc) Note: You are implementing the value that is stored in the __name__ attribute for classes and some built-in types.

Solution:

```
>>> data = "<type 'builtin_function_or_method'>"
>>> import re
>>> patt = 'type \'([A-Za-z_\']+)\''
>>> re.search(patt,data).group(1)
'builtin_function_or_method'
>>>
```

1-14 Processing Dates. In Section 1.2, we gave you the regex pattern that matched the single or double-digit string representations of the months January to September (0?[1-9]). Create the regex that represents the remaining three months in the standard calendar.

Solution:

(0?[1-9])|1[0-2]

1-15. Processing Credit Card Numbers. Also in Section 1.2, we gave you the regex pattern that matched credit card (CC) numbers ([0-9][15,16]). However, this pattern does not allow for hyphens separating blocks of numbers. Create the regex that allows hyphens, but only in the correct of 4-6-5, indicating four digits-hyphen-six digits-hyphen-five digits; and 16-digit CC numbers have a 4-4-4-4 pattern. Rememer to "balloon" the size of the entire string correctly. Extra Credit: There is a standard algorithm for determining whether a CC number is valid. Write some code that not only recognizes a correctly formatted CC number, but also a valid one.

Solution:

[0-9]{4}-[0-9]{6}-[0-9]{5}

[0-9]{4}-[0-9]{4}-[0-9]{4}-[0-9]{4}

Playing with gendata.py. The next set of Exercise (1-16 through 1-27) deal specifically with the data that is generated by gendata.py. Before approaching Exercise 1-17 and 1-18, you might want to do 1-16 and all the regular expressions first.

1-16. Update the code for gendata.py so that the data is written directly to redata.txt rather than output to the screen.

Solution:

Tue May 31 03:28:10 2022::cqoxrh@qybasmx.edu::1653938890-6-7
Wed Jan 22 05:06:30 1997::qgypa@vwdxabhi.edu::853880790-5-8
Wed Mar 14 05:47:12 1979::xdsu@shuuybupl.gov::290209632-4-9
Thu Apr 09 16:09:25 2009::mchxvi@iynvdmmt.edu::1239264565-6-8
Sun Aug 25 19:18:09 1996::aitfioq@pbqptglipouk.edu::840971889-7-12
Sun Mar 14 08:32:13 1982::xdoka@znkrksj.gov::384913933-5-7
Tue Jul 25 17:51:27 2006::tlnzq@rjphsortxtdl.gov::1153821087-5-12
Sun May 08 15:32:01 2022::mejur@iicjqowng.gov::1651995121-5-9

1-17. Determine how many times each day of the week shows up for any incarnation of redata.txt. (Alternatively, you can also count how many times each month of the year was chosen.)

1-18. Ensure that there is no data corruption in redata.txt by confirming that the first integer of the integer field matches the timestamp given at the beginning of each output line.

Create Regular Expressions That:

1-19. Extract the complete timestamps from each line.

Solution:

r"[\w\s:]+[0-9]{4}"

```
>>> patt = r"[\w\s:]+[0-9]{4}"
>>>
>>> m = re.search(patt, data).group()
>>> m
'Wed Mar 14 05:47:12 1979'
>>>
```

1-20. Extract the complete e-mail address from each line.

Solution:

r"\w+@\w+\.(gov|edu|com|org)"

1-21. Extract only the months from the timestamps.

Jan|Feb|Mar|Apr|May|Jun|Jul|Aug|Sep|Oct|Nov|Dec

1-22. Extract only the years from the timestamps.

Solution:

r"\d{4}"

```
>>> patt = r"\d{4}"
>>> m = re.search(patt, data).group()
>>> m
'1979'
>>>
```

1-23. Extract only the time from the timestamps.
r"\d{2}:\d{2}:\d{2}"

1-24 Extract only the login and domain names (both the main domain name and the high-level domain together) from the email address.
Solution:
r"(\w+)@(\w+)\.(gov|edu|com|org)"

```
>>> data
'Wed Mar 14 05:47:12 1979::xdsu@shuuybupl.gov::290209632-4-9'
>>> patt = r"(\w+)@(\w+\.gov|edu|com|org)"
>>> m = re.search(patt, data).groups()
>>> m
('xdsu', 'shuuybupl.gov')
>>>
```

1-25 Extract only the login and domain names (both the main domain name and the high-level domain) from the e-mail address.
Solution:
r"(\w+)@(\w+)\.(gov|edu|com|org)"

```
>>> patt = r"(\w+)@(\w+)\.(gov|edu|com|org)"
>>> m = re.search(patt, data).groups()
>>> m
('xdsu', 'shuuybupl', 'gov')
```

1-26 Replace the e-mail address from each line of data with your email address.
Solution:

```
>>> re.sub(r"(\w+)@(\w+\.gov|edu|com|org)",r"18790166674@163.com", data)
'Wed Mar 14 05:47:12 1979::18790166674@163.com::290209632-4-9'
>>>
```

1-27 Extract the months, days, and years from the timestamps and output them in "Mon, Day, Year" format, iterating over each line only once.
Solution:

```
import re
f = open(r'C:\Users\Administrator\Desktop\redata.txt')
for eachLine in f:
    patt = '\w+\s(\w+)\s(\d+)\s[\d:]+\s(\d+)'
    MDY = re.findall(patt, eachLine)
    print(MDY)
============== RESTART: C:/Users/Administrator/Desktop/1_27.py ==============
[('May', '31', '2022')]
[('Jan', '22', '1997')]
[('Mar', '14', '1979')]
[('Apr', '09', '2009')]
[('Aug', '25', '1996')]
[('Mar', '14', '1982')]
[('Jul', '25', '2006')]
[('May', '08', '2022')]
>>>
```

Processing Telephone Numbers. For Exercise 1-28 and 1-29, recall the regular expression introduced in Section 1.2, which matched telephone numbers but allowed for an optional area code prefix: \d{3}-\d{3}-\d{4}. Update this regular expression so that:

1-28. Area codes (the first set of three-digits and the accompanying hyphen) are optional, that is, your regex should match both 800-555-1212 as well as just 555-1212.

Solution:

(\d{3}-)?\d{3}-\d{4}

1-29. Either parenthesized or hyphenated area codes are supported, not to mention optional; make your regex match 800-555-1212, 555-1212, and also (800)555-1212.

Solution:

'(\((\d{3})\)?|(\d{3}-)?)\d{3}-\d{4}'

```
>>> patt = '(\((\d{3})\)?|(\d{3}-)?)\d{3}-\d{4}'
>>> data = "(800)555-1212"
>>> re.match(patt, data).group()
'(800)555-1212'
>>>
```

The following exercises are too hard for me.
Maybe implementing them in the near future.

Regex Utilities. The final set of exercises make useful utility scripts when processing online data:

1-30. *HTML Generation*. Given a list of links (and optional short description), whether user-provided on command-line, via input from another script, or from a database, generate a Web page (.html) that includes all links as hypertext anchors, which upon viewing in a Web browser, allows users to click those links and visit the corresponding site. If the short description is provided, use that as the hypertext instead of the URL.

1-31 *Tweet Scrub*. Sometimes all you want to see is the pain text of a tweet as posted to the Twitter service by users. Create a function that takes a tweet and optional "meta" flag defaulted False, and then returns a string of the scrubbed tweet, removing all the extraneous information, such as an "RT" notation for "retweet", a leading ., and all"#hashtags". If the meta flag is True, then also return a dict containing the metadata. This can include a key "RT", whose value is a tuple of strings of users who retweeted the message, and/or a key "hashtags" with a tuple of the hashtags. If the values don't exist (empty tuples), then don't even bother creating a key-value entry for them.

1-32 Amazon Screenscraper. Create a script that helps you to keep track of your favorite books and how they're doing on Amazon (or any other online bookseller that tracks book rankings). For example, the Amazon link for any book is of the format, http://amazon.con/dp/ISBN (for example, heep://amazon.com/dp/0132678209). You can then change the domain name to check out the equivalent rankings on Amazon sites in other countries, such as Germany (.de), France (.fr), Japan (.jp), China (.cn), and the UK (.co.uk). Use regular expressions or a markup parser, such as BeautifulSoup, lxml, or html5lib to parse the ranking, and then let the user pass in a command-line argument that specifies whether the output should be in plain text, perhaps for inclusion in an e-mail body, or formatted in HTML for Web consumption.