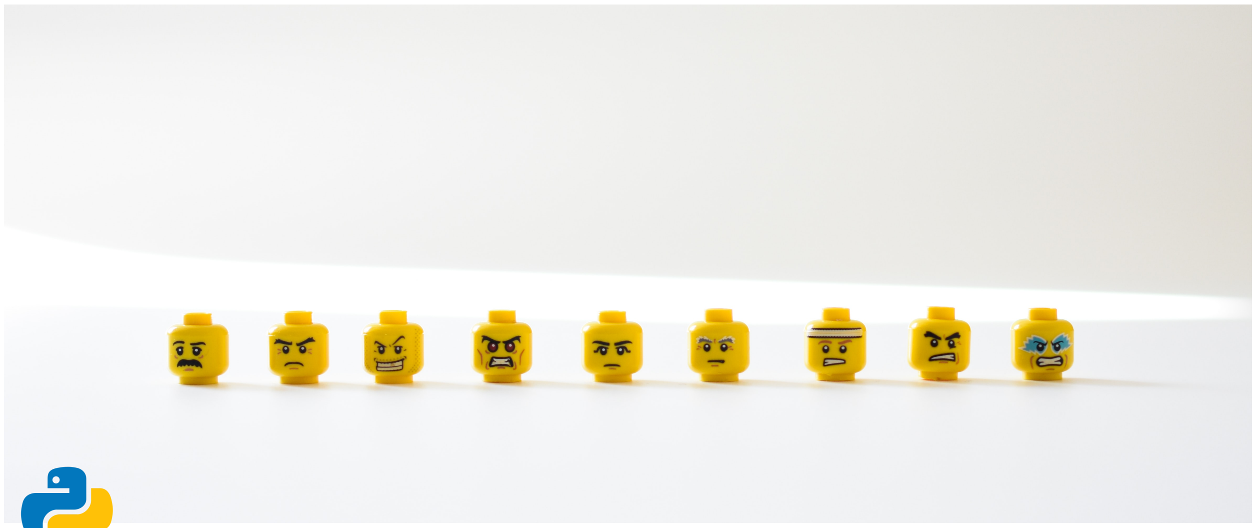

Regular Expressions

Pattern Matching with Regular Expressions

Most of us are familiar with **ctrl + F** and entering the words that you are looking for. Regular Expressions will take a step further and allows you to match a pattern in the text and make necessary changes to the text. Regular expressions are huge time-savers, not just for software users but also for programmers.

Finding Patterns of Text With Regular Expressions

To match a us format phone number formatted like **415-555-4242** would require you to write a multiple lines of codes. The easiest way to do this job is by using regular expressions.

Regular expressions, called *regexes* for short, are descriptions for a pattern in a text. For example, a `\d` in a regex stands for a digit character—that is, any single numeral from 0 to 9.

The regex `\d\d\d-\d\d\d-\d\d\d\d` is used by Python to match the same text pattern of the above mentioned phone number.

But regular expressions can be much more sophisticated. For example, adding a 3 in braces (`{3}`) after a pattern is like saying, “Match this pattern three times.” So the slightly shorter regex `\d{3}-\d{3}-\d{4}` also matches the correct phone number format.

Creating Regex Objects

```
>>> import re
```

💡 Passing a string value representing your regular expression to `re.compile()` returns a `Regex` pattern object (or simply, a `Regex` object).

```
>>> phoneNumRex = re.compile(r'\d\d\d-\d\d\d-\d\d\d\d')
```

Matching Regex Objects

A Regex object’s `search()` method searches the string it is passed for any matches to the regex. The `search()` method will return `None` if the regex pattern is not found in the string. If the pattern *is* found, the `search()` method returns a `Match` object, which have a `group()` method that will return the actual matched text from the searched string.

```
>>> phoneNumRegex = re.compile(r'\d\d\d-\d\d\d-\d\d\d\d')
>>> mo = phoneNumRegex.search('My number is 415-555-4242.')
>>> print('Phone number found: ' + mo.group())
Phone number found: 415-555-4242
```

Here, we pass our desired pattern to `re.compile()` and store the resulting Regex object in `phoneNumRegex`. Then we call `search()` on phoneNumRegex and pass `search()` the string we want to match for during the search. The result of the search gets stored in the variable mo. In this example, we know that our pattern will be found in the string, so we know that a Match object will be returned. Knowing that mo contains a Match object and not the null value None, we can call `group()` on `mo` to return the match. Writing `mo.group()` inside our `print()` function call displays the whole match, 415-555-4242.

```
>>> phoneNumRegex = re.compile(r'(\d\d\d)-(\d\d\d-\d\d\d\d)')
>>> mo = phoneNumRegex.search('My number is 415-555-4242.')
>>> mo.group(1)
'415'
>>> mo.group(2)
'555-4242'
>>> mo.group(0)
'415-555-4242'
>>> mo.group()
'415-555-4242'
```

If you would like to retrieve all the groups at once, use the `groups()` method—note the plural form for the name.

```
>>> mo.groups()
('415', '555-4242')
>>> areaCode, mainNumber = mo.groups()
>>> print(areaCode)
415
>>> print(mainNumber)
555-4242
```

Since `mo.groups()` returns a tuple of multiple values, you can use the multiple-assignment trick to assign each value to a separate variable, as in the previous `areaCode, mainNumber = mo.groups()` line.

The `\(` and `\)` escape characters in the raw string passed to `re.compile()` will match actual parenthesis characters. In regular expressions, the following characters have special meanings:
`. ^ $ * + ? { } [ ] \ | ( )`

If you want to detect these characters as part of your text pattern, you need to escape them with a backslash:
`\. \^ \$ \* \+ \? \{ \} \[ \] \\ \| \( \)`

Matching Multiple Groups with the Pipe

The `|` character is called a *pipe*. You can use it anywhere you want to match one of many expressions. For example, the regular expression `r'Batman|Tina Fey'` will match either `'Batman'` or `'Tina Fey'`.

When *both* Batman and Tina Fey occur in the searched string, the first occurrence of matching text will be returned as the `Match` object.

```
>>> heroRegex = re.compile (r'Batman|Tina Fey')
>>> mo1 = heroRegex.search('Batman and Tina Fey')
>>> mo1.group()
'Batman'

>>> mo2 = heroRegex.search('Tina Fey and Batman')
>>> mo2.group()
'Tina Fey'
```

You can also use the pipe to match one of several patterns as part of your regex. For example, say you wanted to match any of the strings `'Batman', 'Batmobile', 'Batcopter',` and `'Batbat'`. Since all these strings start with `Bat`, it would be nice if you could specify that prefix only once.

```
>>> batRegex = re.compile(r'Bat(man|mobile|copter|bat)')
>>> mo = batRegex.search('Batmobile lost a wheel')
>>> mo.group()
'Batmobile'
>>> mo.group(1)
'mobile'
```

Optional Matching with the Question Mark

Sometimes there is a pattern that you want to match only optionally. That is, the regex should find a match regardless of whether that bit of text is there. The `?` character flags the group that precedes it as an optional part of the pattern.

```
>>> batRegex = re.compile(r'Bat(wo)?man')
>>> mo1 = batRegex.search('The Adventures of Batman')
>>> mo1.group()
'Batman'

>>> mo2 = batRegex.search('The Adventures of Batwoman')
>>> mo2.group()
'Batwoman'
```

Similarly, we can modify our previous code to find phone number in such a way that it identifies the number even if the area code is not present.

```
>>> phoneRegex = re.compile(r'(\d\d\d-)?\d\d\d-\d\d\d\d')
>>> mo1 = phoneRegex.search('My number is 415-555-4242')
>>> mo1.group()
'415-555-4242'

>>> mo2 = phoneRegex.search('My number is 555-4242')
>>> mo2.group()
'555-4242'
```

Matching zero or more with a star

The `*` (called the *star* or *asterisk*) means “match zero or more”—the group that precedes the star can occur any number of times in the text.

```
>>> batRegex = re.compile(r'Bat(wo)*man')
>>> mo1 = batRegex.search('The Adventures of Batman')
>>> mo1.group()
'Batman'

>>> mo2 = batRegex.search('The Adventures of Batwoman')
>>> mo2.group()
'Batwoman'

>>> mo3 = batRegex.search('The Adventures of Batwowowowoman')
>>> mo3.group()
'Batwowowowoman'
```

Matching one or more with the *Plus*

```
>>> batRegex = re.compile(r'Bat(wo)+man')
>>> mo1 = batRegex.search('The Adventures of Batwoman')
>>> mo1.group()
'Batwoman'

>>> mo2 = batRegex.search('The Adventures of Batwowowowoman')
>>> mo2.group()
'Batwowowowoman'

>>> mo3 = batRegex.search('The Adventures of Batman')
>>> mo3 == None
True
```

Matching Specific Repetitions with Braces

If you have a group that you want to repeat a specific number of times, follow the group in your regex with a number in braces. For example, the regex `(Ha){3}` will match the string `'HaHaHa'`, but it will not match `'HaHa'`, since the latter has only two repeats of the `(Ha)` group.

Instead of one number, you can specify a range by writing a minimum, a comma, and a maximum in between the braces. For example, the regex `(Ha){3,5}` will match `'HaHaHa'`, `'HaHaHaHa'`, and `'HaHaHaHaHa'`.

You can also leave out the first or second number in the braces to leave the minimum or maximum unbounded. For example, `(Ha){3,}` will match three or more instances of the `(Ha)` group, while `(Ha){,5}` will match zero to five instances.

The findall() Method

In addition to the `search()` method, `Regex` objects also have a `findall()` method. While `search()` will return a Match object of the *first* matched text in the searched string, the `findall()` method will return the strings of *every* match in the searched string.

On the other hand, `findall()` will not return a Match object but a list of strings—*as long as there are no groups in the regular expression*.

```
>>> phoneNumRegex = re.compile(r'\d\d\d-\d\d\d-\d\d\d\d') # has no groups
>>> phoneNumRegex.findall('Cell: 415-555-9999 Work: 212-555-0000')
['415-555-9999', '212-555-0000']
```

If there *are* groups in the regular expression, then `findall()` will return a list of tuples. Each tuple represents a found match, and its items are the matched strings for each group in the regex.

```
>>> phoneNumRegex = re.compile(r'(\d\d\d)-(\d\d\d)-(\d\d\d\d)') # has groups
>>> phoneNumRegex.findall('Cell: 415-555-9999 Work: 212-555-0000')

[('415', '555', '9999'), ('212', '555', '0000')]
```

Character Classes

In the earlier phone number regex example, you learned that \d could stand for any numeric digit. That is, `\d` is shorthand for the regular expression `(0|1|2|3|4|5|6|7|8|9)`.

Shorthand character class	Represents
`\d`	Any numeric digit from 0 to 9.
`\D`	Any character that is *not* a numeric digit from 0 to 9.
`\w`	Any letter, numeric digit, or the underscore character. (Think of this as matching “word” characters.)
`\W`	Any character that is *not* a letter, numeric digit, or the underscore character.
`\s`	Any space, tab, or newline character. (Think of this as matching “space” characters.)
`\S`	Any character that is *not* a space, tab, or newline.

Shorthand Codes for Common Character Classes

💡 Character classes are nice for shortening regular expressions. The character class `[0-5]` will match only the numbers `0` to `5`; this is much shorter than typing `(0|1|2|3|4|5)`. Note that while `\d` matches digits and `\w` matches digits, letters, and the underscore, there is no shorthand character class that matches only letters.

Though you can use the `[a-zA-Z]` character class

```
>>> xmasRegex = re.compile(r'\d+\s\w+')
>>> xmasRegex.findall('12 drummers, 11 pipers, 10 lords, 9 ladies, 8 maids, 7
swans, 6 geese, 5 rings, 4 birds, 3 hens, 2 doves, 1 partridge')

#['12 drummers', '11 pipers', '10 lords', '9 ladies', '8 maids', '7 swans', '6
# geese', '5 rings', '4 birds', '3 hens', '2 doves', '1 partridge']
```

Making your own character class

There are times when you want to match a set of characters but the shorthand character classes (`\d`, `\w`, `\s`, and so on) are too broad. You can define your own character class using square brackets. For example, the character class `[aeiouAEIOU]` will match any vowel, both lowercase and uppercase.

```
>>> vowelRegex = re.compile(r'[aeiouAEIOU]')
>>> vowelRegex.findall('RoboCop eats baby food. BABY FOOD.')

['o', 'o', 'o', 'e', 'a', 'a', 'o', 'o', 'A', 'O', 'O']
```

You can also include ranges of letters or numbers by using a hyphen. For example, the character class `[a-zA-Z0-9]` will match all lowercase letters, uppercase letters, and numbers.

Note that inside the square brackets, the normal regular expression symbols are not interpreted as such. This means you do not need to escape the `.`, `*`, `?`, or `()` characters with a preceding backslash. For example, the character class `[0-5.]` will match digits 0 to 5 and a period. You do not need to write it as `[0-5\.]`.

By placing a caret character (`^`) just after the character class's opening bracket, you can make a *negative character class*. A negative character class will match all the characters that are *not* in the character class.

```
>>> consonantRegex = re.compile(r'[^aeiouAEIOU]')
```

The Caret and Dollar Sign Characters

You can also use the caret symbol (`^`) at the start of a regex to indicate that a match must occur at the *beginning* of the searched text.

Likewise, you can put a dollar sign (`$`) at the end of the regex to indicate the string must *end* with this regex pattern.

For example, the `r'^Hello'` regular expression string matches strings that begin with `'Hello'`.

The `r'\d$'` regular expression string matches strings that end with a numeric character from 0 to 9.

The `r'^\d+$'` regular expression string matches strings that both begin and end with one or more numeric characters.

The Wildcard Character

The `.` (or *dot*) character in a regular expression is called a *wildcard* and will match any character except for a newline.

```
>>> atRegex = re.compile(r'.at')
>>> atRegex.findall('The cat in the hat sat on the flat mat.')
['cat', 'hat', 'sat', 'lat', 'mat']
```

💡 Remember that the dot character will match just one character, which is why the match for the text `flat` in the previous example matched only `lat`. To match an actual dot, escape the dot with a backslash: `\.`

Sometimes you will want to match everything and anything. For example, say you want to match the string '`First Name:`', followed by any and all text, followed by '`Last Name:`', and then followed by anything again. You can use the dot-star (`.*`) to stand in for that "anything." Remember that the dot character means "any single character except the newline," and the star character means "zero or more of the preceding character."

The dot-star will match everything except a newline. By passing `re.DOTALL` as the second argument to `re.compile()`, you can make the dot character match *all* characters, including the newline character.

```
>>> noNewlineRegex = re.compile('.*')
>>> noNewlineRegex.search('Serve the public trust.\nProtect the innocent.
\nUphold the law.').group()
'Serve the public trust.'


>>> newlineRegex = re.compile('.*', re.DOTALL)
>>> newlineRegex.search('Serve the public trust.\nProtect the innocent.
\nUphold the law.').group()
'Serve the public trust.\nProtect the innocent.\nUphold the law.'
```

Review of Regex Symbols

This chapter covered a lot of notation on *regex*, so here's a quick review of what you learned about basic regular expression syntax:

- The `?` matches zero or one of the preceding group.
- The `*` matches zero or more of the preceding group.
- The `+` matches one or more of the preceding group.
- The `{n}` matches exactly *n* of the preceding group.
- The `{n,}` matches *n* or more of the preceding group.
- The `{,m}` matches 0 to *m* of the preceding group.
- The `{n,m}` matches at least *n* and at most *m* of the preceding group.
- `{n,m}?` or `*?` or `+?` performs a non-greedy match of the preceding group.
- `^spam` means the string must begin with *spam*.
- `spam$` means the string must end with *spam*.
- The `.` matches any character, except newline characters.
- `\d`, `\w`, and `\s` match a digit, word, or space character, respectively.
- `\D`, `\W`, and `\S` match anything except a digit, word, or space character, respectively.

- `[abc]` matches any character between the brackets (such as *a*, *b*, or *c*).
- `[^abc]` matches any character that isn't between the brackets.

Case-Insensitive Matching

Normally, regular expressions match text with the exact casing you specify. For example, the following regexes match completely different strings:

```
>>> regex1 = re.compile('RoboCop')
>>> regex2 = re.compile('ROBOCOP')
>>> regex3 = re.compile('robOcop')
>>> regex4 = re.compile('RobocOp')
```

To make your regex case-insensitive, you can pass `re.IGNORECASE` or `re.I` as a second argument to `re.compile()`.

```
>>> robocop = re.compile(r'robocop', re.I)
>>> robocop.search('RoboCop is part man, part machine, all cop.').group()

'RoboCop'
```

Substituting Strings with the sub() Method

The `sub()` method for `Regex` objects is passed two arguments. The first argument is a string to replace any matches.

```
>>> namesRegex = re.compile(r'Agent \w+')
>>> namesRegex.sub('CENSORED', 'Agent Alice gave the secret documents to Agent Bob.')

'CENSORED gave the secret documents to CENSORED.'
```

Using VERBOSE with regular expression

REs of moderate complexity can become lengthy collections of backslashes, parentheses, and metacharacters, making them difficult to read and understand.

For such REs, specifying the `re.VERBOSE` flag when compiling the regular expression can be helpful, because it allows you to format the regular expression more clearly.

The `re.VERBOSE` flag has several effects. Whitespace in the regular expression that *isn't* inside a character class is ignored. This means that an expression such as `dog | cat` is equivalent to the less readable `dog|cat`, but `[a b]` will still match the characters "a", "b", or a space. In addition, you can also put comments inside a RE; comments extend from a "#" character to the next newline. When used with triple-quoted strings, this enables REs to be formatted more neatly:

```
pat = re.compile(r"""
 \s*                 # Skip leading whitespace
 (?P<header>[^:]+)   # Header name
 \s* :               # Whitespace, and a colon
 (?P<value>.*?)      # The header's value -- *? used to
                     # lose the following trailing whitespace
 \s*$                # Trailing whitespace to end-of-line
""", re.VERBOSE)
```

This is far more readable than,

`pat = re.compile(r"\s*(?P<header>[^:]+)\s*:(?P<value>.*?)\s*$")`

🕹️Project - Finding Emails

```
import pyperclip, re



# Create email regex.
emailRegex = re.compile(r'''(
    [a-zA-Z0-9._%+-]+
    @
    [a-zA-Z0-9.-]+
    (\.[a-zA-Z]{2,4})
    )''', re.VERBOSE)

text = str(pyperclip.paste())
```

```
matches = []

for groups in emailRegex.findall(text):
       matches.append(groups[0])

if len(matches) > 0:
    pyperclip.copy('\n'.join(matches))
    print('Copied to clipboard:')
    print('\n'.join(matches))
else:
    print('No email addresses found.')
```

go to: https://nostarch.com/contactus/

Copy the entire page by clicking ctrl+A. Then run the program to get desired output.