6 Common Problems

Regular expressions are a powerful tool for some applications, but in some ways their behaviour isn't intuitive and at times they don't behave the way you may expect them to. This section will point out some of the most common pitfalls.

6.1 Use String Methods

Sometimes using the `re` module is a mistake. If you're matching a fixed string, or a single character class, and you're not using any `re` features such as the `IGNORECASE` flag, then the full power of regular expressions may not be required. Strings have several methods for performing operations with fixed strings and they're usually much faster, because the implementation is a single small C loop that's been optimized for the purpose, instead of the large, more generalized regular expression engine.

One example might be replacing a single fixed string with another one; for example, you might replace `word` with `deed`. `re.sub()` seems like the function to use for this, but consider the `replace()` method. Note that `replace()` will also replace `word` inside words, turning `swordfish` into `sdeedfish`, but the naive RE `word` would have done that, too. (To avoid performing the substitution on parts of words, the pattern would have to be `\bword\b`, in order to require that `word` have a word boundary on either side. This takes the job beyond `replace()`'s abilities.)

Another common task is deleting every occurrence of a single character from a string or replacing it with another single character. You might do this with something like `re.sub('\n', ' ', S)`, but `translate()` is capable of doing both tasks and will be faster than any regular expression operation can be.

In short, before turning to the `re` module, consider whether your problem can be solved with a faster and simpler string method.

6.2 match() versus search()

The `match()` function only checks if the RE matches at the beginning of the string while `search()` will scan forward through the string for a match. It's important to keep this distinction in mind. Remember, `match()` will only report a successful match which will start at 0; if the match wouldn't start at zero, `match()` will *not* report it.

```
>>> print(re.match('super', 'superstition').span())
(0, 5)
>>> print(re.match('super', 'insuperable'))
None
```

On the other hand, `search()` will scan forward through the string, reporting the first match it finds.

```
>>> print(re.search('super', 'superstition').span())
(0, 5)
>>> print(re.search('super', 'insuperable').span())
(2, 7)
```

Sometimes you'll be tempted to keep using `re.match()`, and just add `.*` to the front of your RE. Resist this temptation and use `re.search()` instead. The regular expression compiler does some analysis of REs in order to speed up the process of looking for a match. One such analysis figures out what the first character of a match must be; for example, a pattern starting with `Crow` must match starting with a `'C'`. The analysis lets the engine quickly scan through the string looking for the starting character, only trying the full match if a `'C'` is found.

Adding `.*` defeats this optimization, requiring scanning to the end of the string and then backtracking to find a match for the rest of the RE. Use `re.search()` instead.