POWERING POTENTIAL

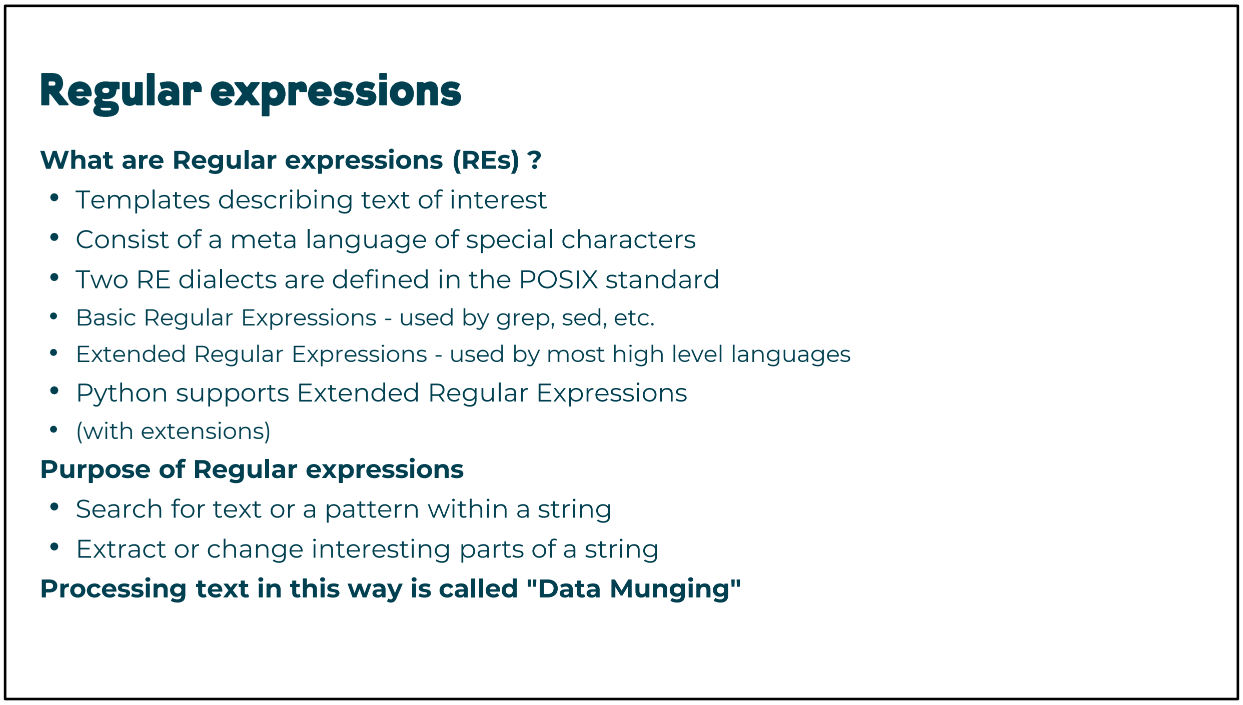
Python 3 Programming

Regular Expressions

REGULAR EXPRESSIONS

Contents

- Python Regular Expressions
- Elementary extended RE meta-characters
- Regular expression objects
- Regular expression substitution
- Matching alternatives
- Anchors
- Class shortcuts
- Repeat quantifiers
- Parentheses groups
- Back-references



Regular expressions

What are Regular expressions (REs) ?

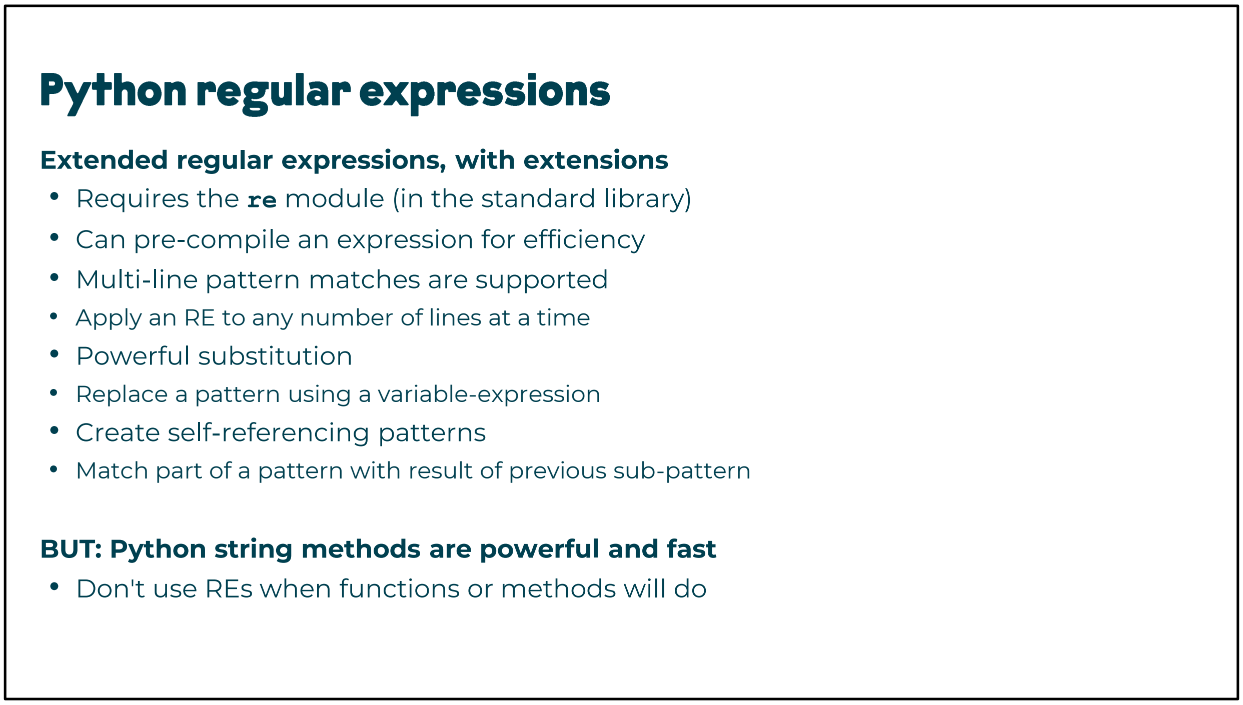
- Templates describing text of interest
- Consist of a meta language of special characters
- Two RE dialects are defined in the POSIX standard
- Basic Regular Expressions - used by grep, sed, etc.
- Extended Regular Expressions - used by most high level languages
- Python supports Extended Regular Expressions
- (with extensions)

Purpose of Regular expressions

- Search for text or a pattern within a string
- Extract or change interesting parts of a string

Processing text in this way is called "Data Munging"

Regular Expressions are commonly used in UNIX command-line tools, and are also present in .Net. Most programming languages support them in one form or another, either built-in or through an external library.
They have a reputation for being difficult to understand, but they are well worth the effort to learn.



Python regular expressions

Extended regular expressions, with extensions

- Requires the `re` module (in the standard library)
- Can pre-compile an expression for efficiency
- Multi-line pattern matches are supported
  - Apply an RE to any number of lines at a time
- Powerful substitution
  - Replace a pattern using a variable-expression
- Create self-referencing patterns
  - Match part of a pattern with result of previous sub-pattern

BUT: Python string methods are powerful and fast

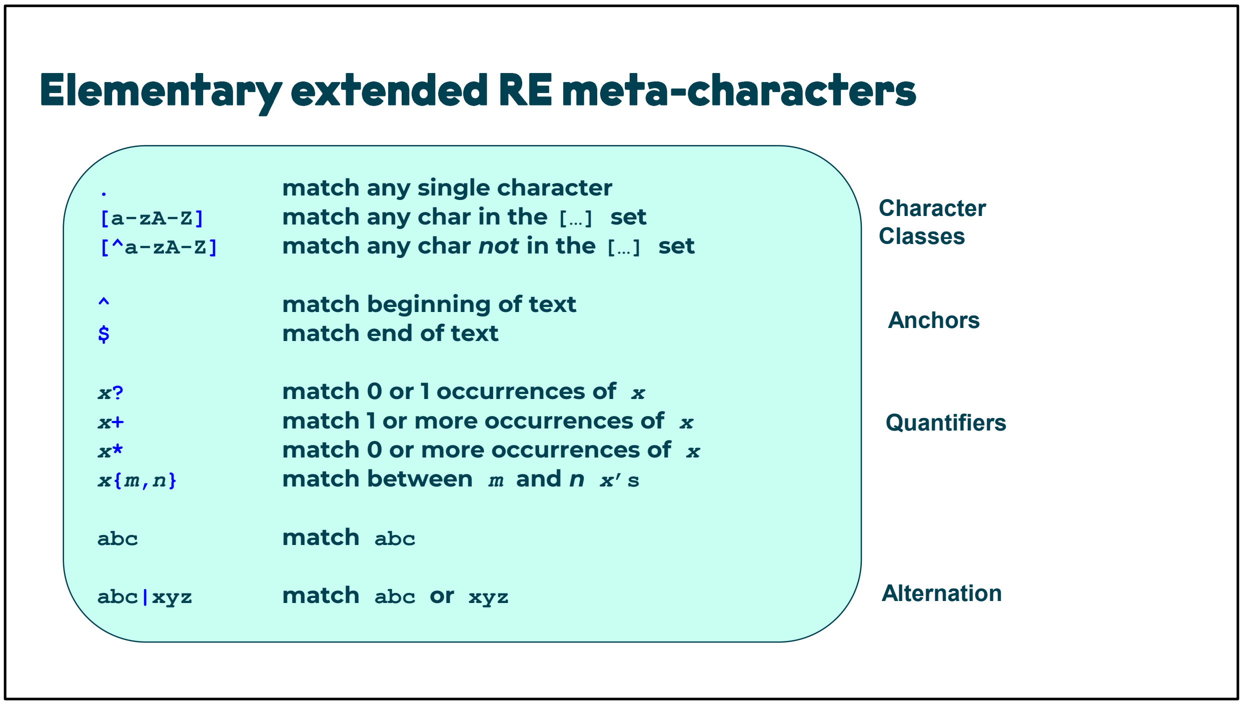
- Don't use REs when functions or methods will do

Python regular expressions are very powerful, yet are easy to learn if you are familiar with tools like grep, vi, sed and awk. Python uses Extended Regular Expressions (ERE), whereas many UNIX tools, like grep, use Basic Regular Expression (BRE) syntax. The most noticeable difference is that BREs need to 'escape' (backslash) parentheses and braces, but thankfully we do not need that in Python.
Here is an overview of the most important differences between grep/sed/awk and Python regular expressions:

Python REs can be applied across multiple lines, so you can match a group of lines that match specific large patterns, without having to keep track of state yourself.

Python substitution expressions allow you to use variables in the search text, including dictionary and list lookups using parts of the pattern just being matched. You can even replace a pattern by the result of a function or subroutine, being called with selected parts of the pattern being replaced.

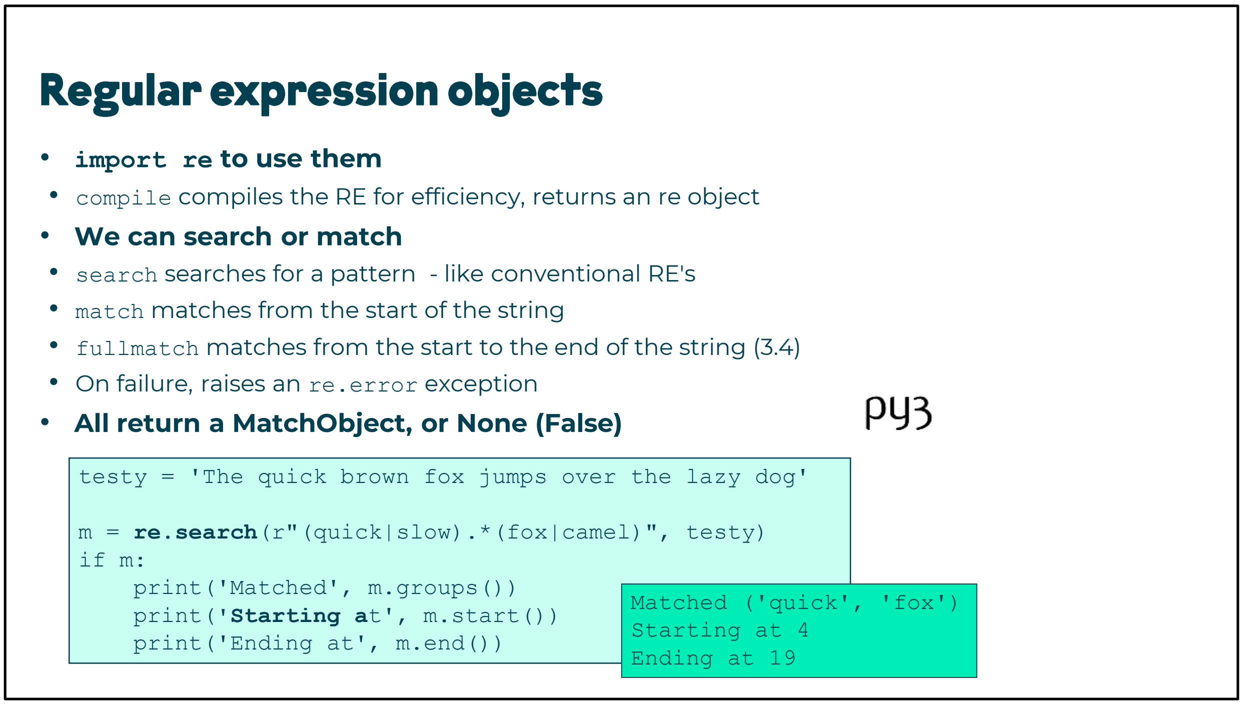
Python patterns can be self-referencing: you can build a pattern of multiple parts, and then say ‘between these two expressions, I want to match whatever I am currently matching at sub-pattern x'.



Elementary extended RE meta-characters

`.`	match any single character	Character Classes
`[a-zA-Z]`	match any char in the `[...]` set	
`[^a-zA-Z]`	match any char *not* in the `[...]` set	
`^`	match beginning of text	Anchors
`$`	match end of text	
x`?`	match 0 or 1 occurrences of *x*	Quantifiers
x`+`	match 1 or more occurrences of *x*	
x`*`	match 0 or more occurrences of *x*	
x`{`*m*`,`*n*`}`	match between *m* and *n* *x*'s	
`abc`	match `abc`	
`abc|xyz`	match `abc` or `xyz`	Alternation

The examples listed above cover the most common regular expression meta-characters. If you are new to regular expressions, this is a good table to remember.
The expression Character Class is introduced here. A Character Class may be specified in the 'traditional' way as show, using the square brackets [ ].
Note that meta-characters used inside [...] are different to those used outside. Escaped meta-characters (those prefixed with /) are literals, as are meta-characters inside [].



Regular expression objects

- **`import re` to use them**
  - `compile` compiles the RE for efficiency, returns an re object
- **We can search or match**
  - `search` searches for a pattern - like conventional RE's
  - `match` matches from the start of the string
  - `fullmatch` matches from the start to the end of the string (3.4)
  - On failure, raises an `re.error` exception
- **All return a MatchObject, or None (False)**

py3

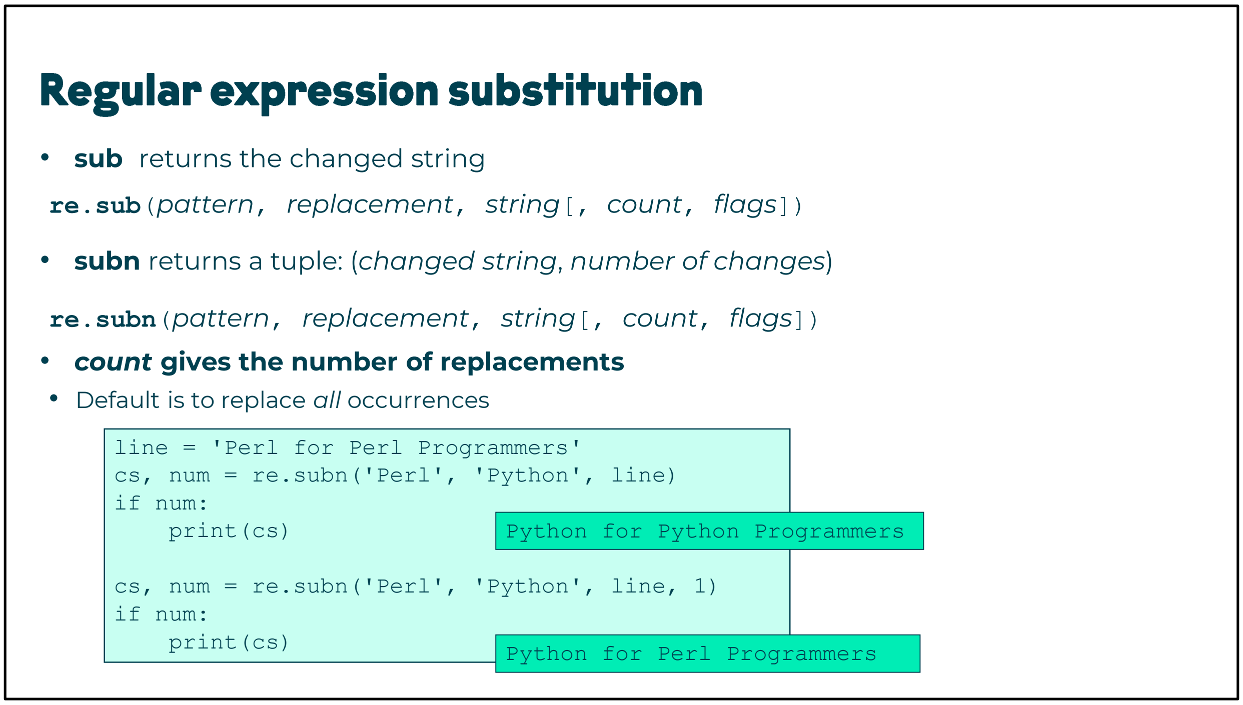
```
testy = 'The quick brown fox jumps over the lazy dog'

m = re.search(r"(quick|slow).*(fox|camel)", testy)
if m:
    print('Matched', m.groups())
    print('Starting at', m.start())
    print('Ending at', m.end())
```

```
Matched ('quick', 'fox')
Starting at 4
Ending at 19
```

Importing the `re` module allows methods on the re class to be called, with search and match being the most common. The `fullmatch` method was introduced at Python 3.4. They all return an object of class `MatchObject`.
The group in parentheses is also known as a *capturing parentheses group*. Text inside parentheses may be referred to later in the RE as a back-reference. A useful method is `groups()`, which returns a tuple containing the matched text, and may be used like back-references. Other methods include `start()` and `end()`, which return the positions of the match, and `group()` which returns the matched string. There are several `MatchObject` attributes, including `re`, which gives the original regular expression, and `string` which gives the original input.
If the search (or match) failed to find the pattern then the empty object None is returned, which is false in Boolean context and a `re.error` exception (enhanced in Python 3.5) is raised.
Python RE syntax is like that used by lower level language libraries, such as the GNU C regex package.
We can compile our REs for efficiency, for example:

```
      reobj = re.compile (r"([Ii]).*(\1)")
```

```
    for line in file:
        m = reobj.match(line)
        if m:
            print(m.string[m.start():m.end()])
```
Notice the use of raw strings (r"..."), these mean we do not have to escape (\) special characters like brackets and braces – particularly useful with Regular Expressions.



Regular expression substitution

- **sub** returns the changed string

`re.sub``(`*pattern*`, `*replacement*`, `*string*`[, `*count*`, `*flags*`])`

- **subn** returns a tuple: (*changed string*, *number of changes*)

`re.subn``(`*pattern*`, `*replacement*`, `*string*`[, `*count*`, `*flags*`])`

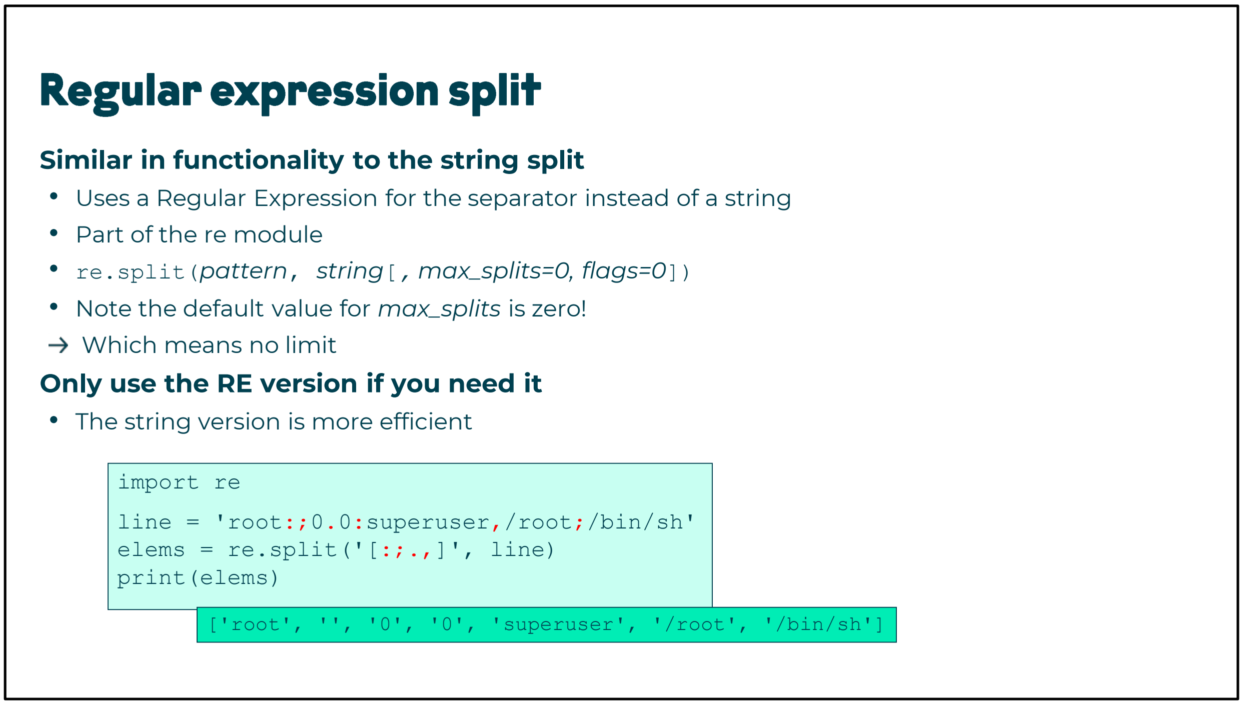
- ***count* gives the number of replacements**
  - Default is to replace *all* occurrences

```
line = 'Perl for Perl Programmers'
cs, num = re.subn('Perl', 'Python', line)
if num:
    print(cs)

cs, num = re.subn('Perl', 'Python', line, 1)
if num:
    print(cs)
```

```
Python for Python Programmers
```

```
Python for Perl Programmers
```

Substitution is reminiscent of awk, in that we have a couple of method calls. sub is used where we just want the new string, whereas subn returns both the altered string and the number of matches. Both perform global substitutions from the left of the string.
There are other useful re functions, for example split, which is shown on the next slide.
We have also shown a compiled Regular Expression object. Compiling the RE makes for more efficient code when the same pattern is used many times, for example in a loop. Methods like match, fullmatch, findall, search, split, sub, and subn may be called on these objects.



Regular expression split

Similar in functionality to the string split

- Uses a Regular Expression for the separator instead of a string
- Part of the re module
- `re.split(`*pattern*`,` *string*`[,` *max_splits=0, flags=0*`])`
- Note the default value for *max_splits* is zero!

→ Which means no limit

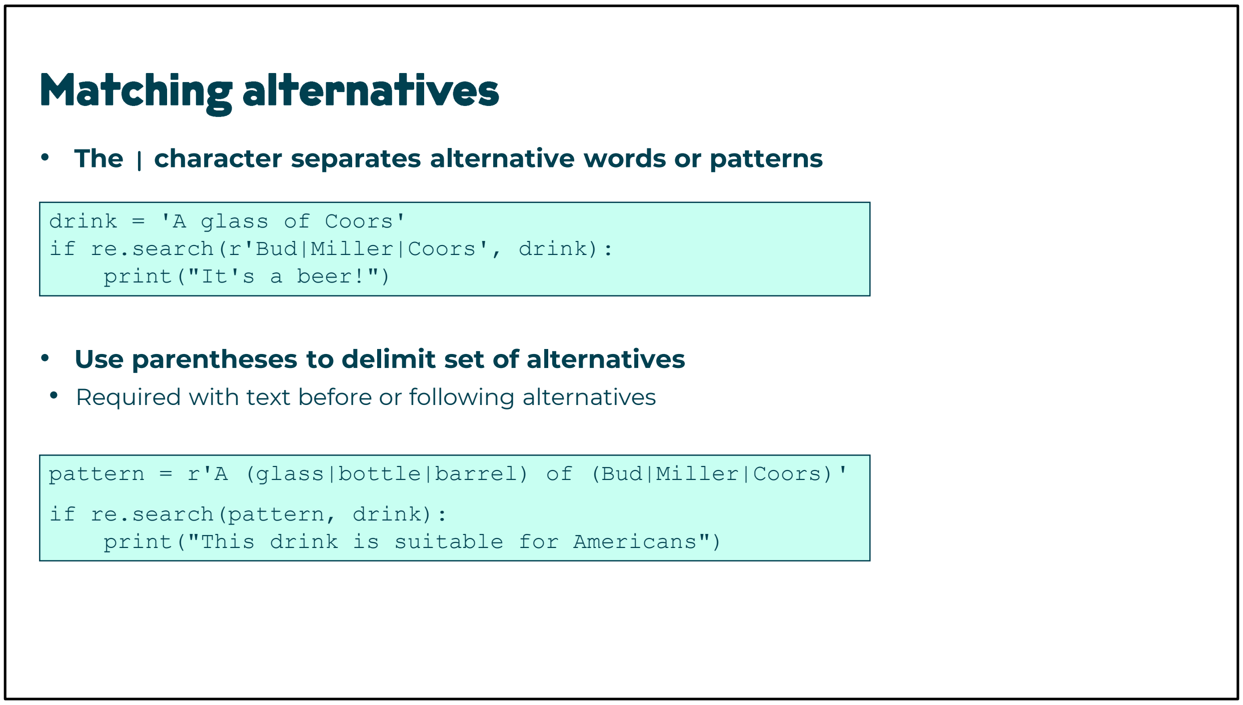
Only use the RE version if you need it

- The string version is more efficient

```
import re

line = 'root:;0.0:superuser,/root;/bin/sh'
elems = re.split('[:;.,]', line)
print(elems)
```

```
['root', '', '0', '0', 'superuser', '/root', '/bin/sh']
```

The **`re`** module's version of **`split`** enables a regular expression to be used to describe the field delimiter (this is much like `split` in Perl and PHP).
The optional parameter *flags* was added at 3.1 (see later).



Matching alternatives

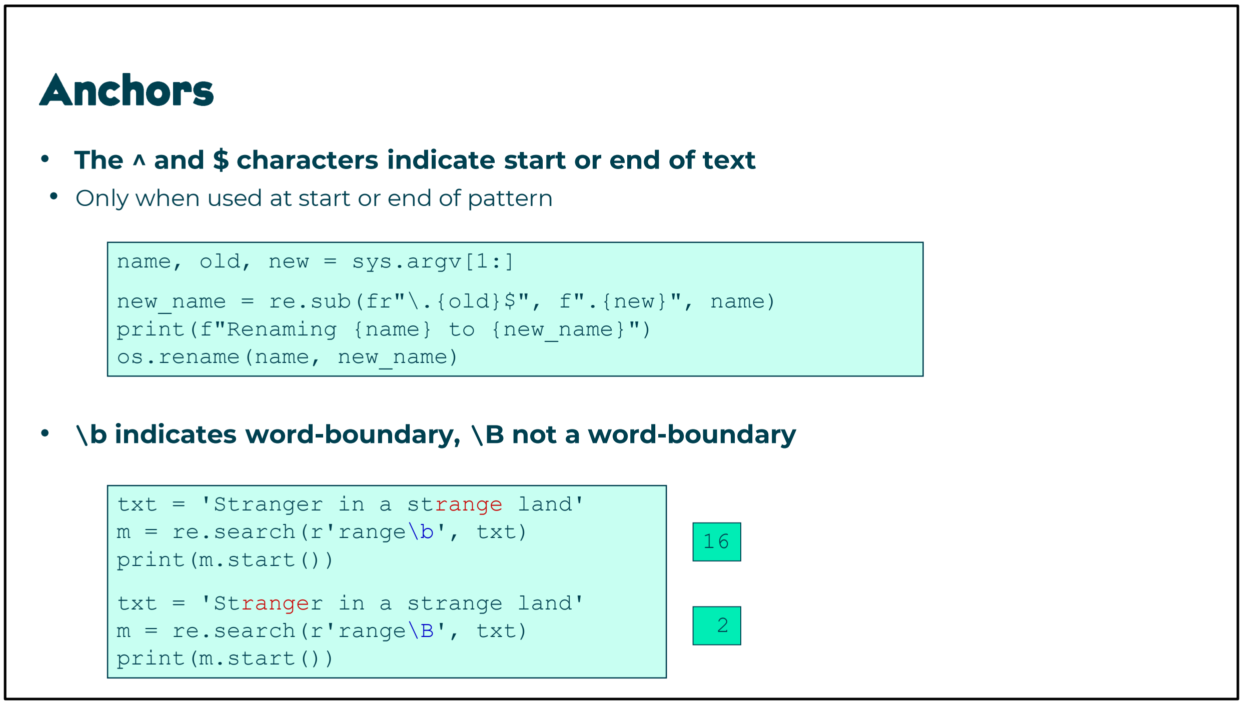
- **The `|` character separates alternative words or patterns**

```
drink = 'A glass of Coors'
if re.search(r'Bud|Miller|Coors', drink):
    print("It's a beer!")
```

- **Use parentheses to delimit set of alternatives**
  - Required with text before or following alternatives

```
pattern = r'A (glass|bottle|barrel) of (Bud|Miller|Coors)'

if re.search(pattern, drink):
    print("This drink is suitable for Americans")
```

The second example above can match nine different possibilities. A second effect of the parenthesis-notation is that they capture the matched text in the groups list, which we see later.



Anchors

- **The ^ and $ characters indicate start or end of text**
  - Only when used at start or end of pattern

```
name, old, new = sys.argv[1:]
new_name = re.sub(fr"\.{old}$", f".{new}", name)
print(f"Renaming {name} to {new_name}")
os.rename(name, new_name)
```

- **\b indicates word-boundary, \B not a word-boundary**

```
txt = 'Stranger in a strange land'
m = re.search(r'range\b', txt)
print(m.start())
```

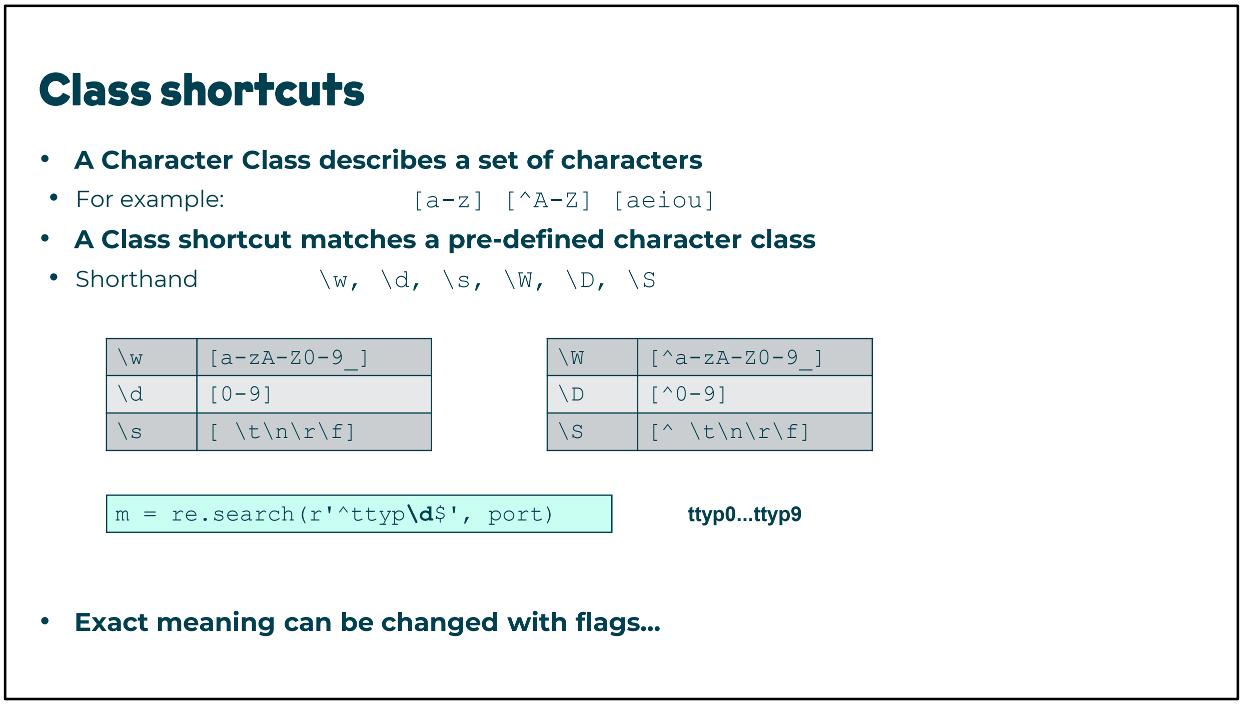
```
16
```

```
txt = 'Stranger in a strange land'
m = re.search(r'range\B', txt)
print(m.start())
```

```
2
```

The $ anchor is special: if you use /00$/, it will match either two zeroes at the end of the search text, or two zeroes followed by a newline at the end of a search text. (it automatically ignores a new-line at the end of a line).
When the ^ character is used anywhere except at the start of a pattern, it indicates a normal ^ character.
When you have a search text that contains multiple lines, the ^ and $ anchors apply to the whole of the text. If you use the m flag (see later), they will be applied to each individual line within the search text. For single-line matches, this is of no importance, but for multi-line matches genuine start of text can be marked with \A, and end of text with \Z.
In addition to the start and end of line anchors, we have the word anchor, \b. It indicates a word boundary (either the beginning or the end of a word), but, like ^ and $, does not take up any space. Exactly what constitutes a word boundary is, however, not always intuitive when it comes to apostrophes. The non-word boundary anchor, \B is used when we explicitly want the text imbedded in another word.
Beware! When used in square brackets (a character class) \b

means a single back-space character!



Class shortcuts

- **A Character Class describes a set of characters**
  - For example: `[a-z] [^A-Z] [aeiou]`
- **A Class shortcut matches a pre-defined character class**
  - Shorthand `\w, \d, \s, \W, \D, \S`

`\w`	`[a-zA-Z0-9_]`
`\d`	`[0-9]`
`\s`	`[ \t\n\r\f]`

`\W`	`[^a-zA-Z0-9_]`
`\D`	`[^0-9]`
`\S`	`[^ \t\n\r\f]`

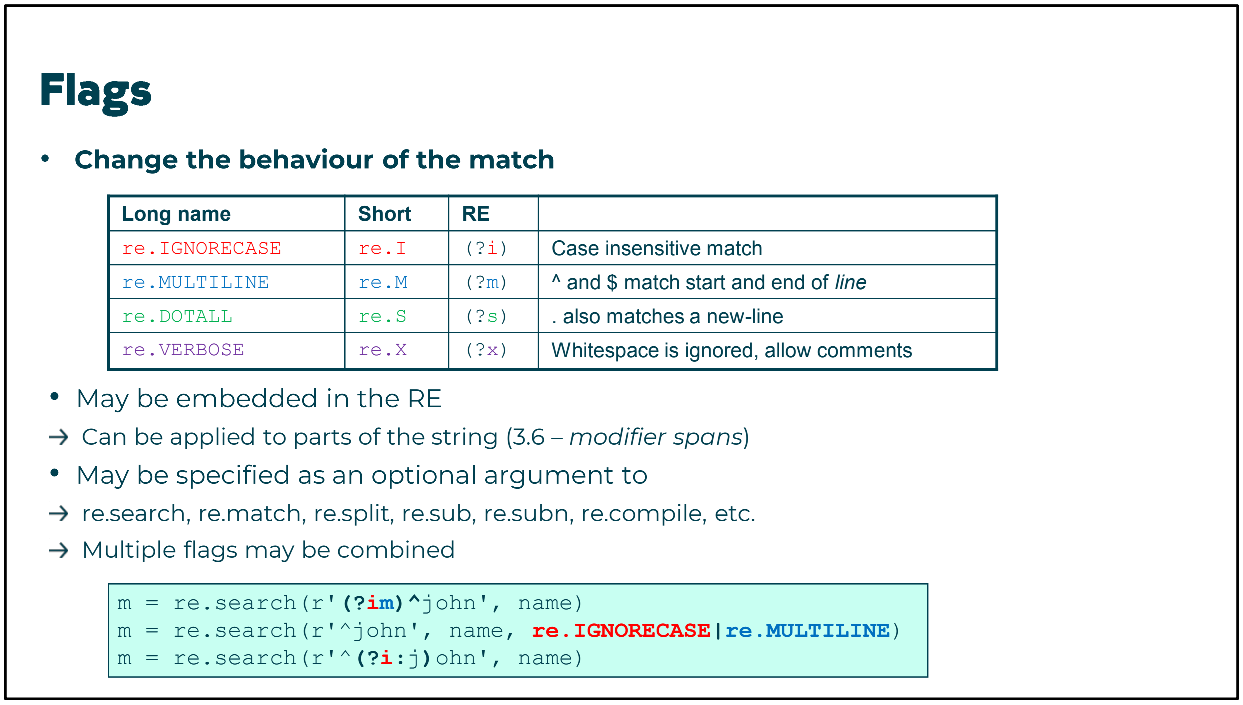
```
m = re.search(r'^ttyp\d$', port)
```

ttyp0...ttyp9

- **Exact meaning can be changed with flags...**

The character classes work as in sed and awk, however the typed character classes may also be used, so [\da-fA-F] matches any hexadecimal digit. Note that the character classes may vary with the locale, depending on which flags are set.
There are other \ functions, for example:

\G	Continue where previous match left off



Flags

- **Change the behaviour of the match**

Long name	Short	RE	
`re.IGNORECASE`	`re.I`	`(?i)`	Case insensitive match
`re.MULTILINE`	`re.M`	`(?m)`	^ and $ match start and end of *line*
`re.DOTALL`	`re.S`	`(?s)`	. also matches a new-line
`re.VERBOSE`	`re.X`	`(?x)`	Whitespace is ignored, allow comments

- May be embedded in the RE

→ Can be applied to parts of the string (3.6 – *modifier spans*)

- May be specified as an optional argument to

→ re.search, re.match, re.split, re.sub, re.subn, re.compile, etc.

→ Multiple flags may be combined

```
m = re.search(r'(?im)^john', name)
m = re.search(r'^john', name, re.IGNORECASE|re.MULTILINE)
m = re.search(r'^(?i:j)ohn', name)
```

It would be tempting to state that the short names are the initial letter of the long name, and that the RE syntax is just the short name in lowercase. You can see that this is not the case, the S and X flags are there for compatibility with other RE engines.
The optional *flags* parameter was added to the `re` methods at Python 3.1.
The first two examples combine the IGNORECASE and MULTILINE flags. They look for 'john' in any case at the start of the text *or* immediately after a new-line character. The third example is a *modifier span* and only applies the `i` (ignore case) to the letter j – so John or john but not JOHN.
When embedded in the RE the single characters can be in any order. When using the re module attribute flags, they are combined with a binary OR |, also in any order. The flags and embedded attributes may be mixed, but that might make the RE even more confusing.
There are two additional flags not shown on the slide. From Python 3.5, they are deprecated, and from 3.6 are only supported for byte

Long name	Short	RE	
`re.ASCII`	`re.A`	`(?a)`	Class shortcuts do not include Unicode
`re.LOCALE`	`re.L`	`(?L)`	Class shortcuts are locale sensitive

SUMMARY

- **Regular expressions are used by many programs**
- **Regular expressions create a MatchObject on match**
- **Many commands:**

`re.search`	- find a pattern somewhere in the string
`re.match`	- match from the start of the string
`re.fullmatch`	- match from start to end of string (3.4)
`re.sub`	- substitute the pattern, returning the new string
`re.subn`	- substitute the pattern, returning the new string and a count of substitutions

Class shortcuts		**. \w \d \s \W \D \S**
Alternatives		
	characters	[ ]
	strings	|
Repeat qualifiers		? * + {*m*,*n*}