```
In [1]: import codecs
        import unicodedata
        with codecs.open("faust.txt","r","utf-8") as stream: text = stream.read()
```

```
In [2]: # !sudo locale-gen de_DE.UTF-8
```

```
In [3]: import locale
        locale.setlocale(locale.LC_ALL,'de_DE.utf8')
        # C, en_US.utf8, ...

Out[3]: 'de_DE.utf8'
```

Basic Searching and Matching

The `re` (regular expression) module contains all the functions we are talking about here.

Regular expressions are powerful tools for searching for strings and patterns.

They are the basis of the command line `fgrep`, `grep`, and `egrep` tools (the `re` in those names stands for "regular expression").

Internally, the query is converted into a finite state automaton, and that automaton is then matched.

```
In [4]: import re
```

There are two basic operations, `search` and `match`. The first searches for a regular expression anywhere, the second requires the match to start at the beginning.

A *match* is indicated by returning a regular expression object (this behaves like a boolean `True`), and a failed match is indicated by returning None.

```
In [5]: re.search('cheese','the cheese and the bread')

Out[5]: <_sre.SRE_Match at 0x40ab988>
```

```
In [6]: re.search('butter','the cheese and the bread')
```

```
In [7]: re.match('cheese','the cheese and the bread')
```

```
In [8]: re.match('the','the cheese and the bread')

Out[8]: <_sre.SRE_Match at 0x40aba58>
```

Matches are case-sensitive by default.

```
In [9]: re.search('THE','the cheese and the bread')
```

But we can make matches case insensitive with the `re.I` flag.

```
In [15]: re.search('THE','the cheese and the bread',re.I)

Out[15]: <_sre.SRE_Match at 0x40abd30>
```

We can also incorporate this flag directly into the query.

```
In [16]: re.search('THE(?i)','the cheese and the bread')

Out[16]: <_sre.SRE_Match at 0x40abd98>
```

A third important operation is `sub` and its variant `subn`.

```
In [10]: re.sub('cheese','butter','bread and cheese')

Out[10]: 'bread and butter'
```

```
In [11]: re.subn('cheese','butter','bread and cheese')

Out[11]: ('bread and butter', 1)
```

Also, we can find multiple matches with `findall`.

```
In [12]: re.findall('spam','spam, spam, ham, and spam')

Out[12]: ['spam', 'spam', 'spam']
```

Finally, we can also split.

```
In [13]: re.split(' ','the quick brown fox')

Out[13]: ['the', 'quick', 'brown', 'fox']
```

Flags

Regular expression operations also take a number of flags that affect the operation:

- `re.I` - ignore case
- `re.L` - locale-dependent matches
- `re.M` - multiline (changes meaning of $ and ^)
- `re.S` - dot matches all characters (usually doesn't match `\n`)
- `re.X` - verbose regular expressions (whitespace is ignored and allows comments)
- `re.U` - unicode-dependent matches (changes interpretation of digits etc)

You can also specify these with syntax like `(?iu)` inside the expression.

```
In [19]: re.findall(r'THE','the cat in the hat',re.I)

Out[19]: ['the', 'the']
```

```
In [20]: re.findall(r'THE(?i)','the cat in the hat')

Out[20]: ['the', 'the']
```

Match Objects

The match object gives additional information about the match. It contains "groups"; group 0 refers to the entire match (we'll see how to define other groups later).

```
In [21]: g = re.search('cheese','the cheese and the bread')
         g

Out[21]: <_sre.SRE_Match at 0x3b8e098>
```

```
In [22]: g.group(0)

Out[22]: 'cheese'
```

```
In [23]: g.start(0),g.end(0)

Out[23]: (4, 10)
```

Precompiled Regular Expressions

Regular expression matching is a two step process:

- the expression string is compiled (into a finite automaton)
- the automaton is executed

Compilation can be costly, so you can separate it from matching and substitution.

```
In [24]: obj = re.compile('cheese')
         obj

Out[24]: re.compile(r'cheese')
```

```
In [25]: obj.search('bread and cheese')

Out[25]: <_sre.SRE_Match at 0x3b8e100>
```

```
In [26]: obj.match('bread and cheese')
In [27]: obj.sub('butter','bread and cheese')

Out[27]: 'bread and butter'
```

Raw Strings

Regular expressions frequently involve backslash characters (\), and sometimes also single or double quotes. For this, there are several convenient quoting conventions:

- `r"abc"` - raw string
- `"""a"bc"""` - triple quoted
- `r"""a"bc"""` - triple quoted raw
- `ur"""a"bc"""` - triple quoted raw unicode string

```
In [28]: print 'a\bc'
         print r'a\bc'
         print "a\"b\"c"
         print r"""a\"b\"c"""
         print ur"""a\"b\"c"""

         ac
         a\bc
         a"b"c
         a\"b\"c
         a\"b\"c
```

```
In [29]: re.search(r'\w+','the bread and the cheese').group(0)

Out[29]: 'the'
```

```
In [30]: re.search(ur'\w+',u'Brot und Käse').group(0)

Out[30]: u'Brot'
```

Unicode Matching

Be careful when matching Unicode in Python 2.x, since you can write either or both the regular expression and the target as `str` or `unicode`. If you aren't consistent, the matches will just fail.

Furthermore, matching UTF-8 encodings stored in `str` won't work right.

```
In [31]: re.search(ur'Käse',u'Der Käse und das Brot.')

Out[31]: <_sre.SRE_Match at 0x3b8e2a0>
```

```
In [32]: re.search('Käse',u'Der Käse und das Brot.')
In [33]: re.search(ur'Käse','Der Käse und das Brot.')
```

```
In [34]: re.search('Käse','Der Käse und das Brot.')

Out[34]: <_sre.SRE_Match at 0x3b8e308>
```

Even if both strings are Unicode, you still have to worry about normalization.

```
In [35]: s = unicodedata.normalize('NFD',u'Käse')
         print "(%s)"%s
         re.search(s,'Der Käse und das Brot')

          (Käse)
```

```
In [36]: def normalizing_search(regex,s):
             regex = unicodedata.normalize('NFC',regex)
             s = unicodedata.normalize('NFC',s)
             return re.search(regex,s)
```

```
In [37]: normalizing_search(s,u'Der Käse und das Brot')

Out[37]: <_sre.SRE_Match at 0x3b8e370>
```

Basic Regular Expression Syntax

There are a number of standard syntactic elements:

- `.` matches a single character (any character)
- `x*` matches 0 or more x
- `x+` matches 1 or more x
- `x?` matches 0 or 1 x
- ^ and $ match at the beginning and end of a line, respectively
- `\x` suppresses the special meaning of character x
- `(xyz)` matches `xyz` and treats it as a unit for the purpose of operators (it also defines a group)
- `x|y` matches x or y
- `[abcA-Z]` matches any one character in the set a, b, c, or in the range A through Z
- `[^abc]` matches any character other than a, b, or c

```
In [38]: re.findall('c.t','the cat on the cot')

Out[38]: ['cat', 'cot']
```

```
In [39]: re.findall('we*t','wet cowtippers tweet frequently')

Out[39]: ['wet', 'wt', 'weet']
```

```
In [40]: re.findall('we+t','wet cowtippers tweet frequently')

Out[40]: ['wet', 'weet']
```

```
In [41]: re.findall('we?t','wet cowtippers tweet frequently')

Out[41]: ['wet', 'wt']
```

There is actually a generalization of the *-like operators, where you can specify the exact number of repetitions with syntax like `{3,7}`.

```
In [42]: re.findall('[ew]t','wet cowtippers tweet frequently')

Out[42]: ['et', 'wt', 'et']
```

```
In [43]: print re.findall(r'\^\.\^','this ^.^ is a Japanese smiley, ^_^')
         print re.findall(r'\^.\^','this ^.^ is a Japanese smiley, ^_^')

         ['^.^']
         ['^.^', '^_^']
```

```
In [44]: print re.findall(r'w','wet cowtippers tweet frequently')
         print re.findall(r'^w','wet cowtippers tweet frequently')

         ['w', 'w', 'w']
         ['w']
```

```
In [45]: print re.findall(r'(tweet|twit)','wet cowtippers tweet frequently, but are twits')

         ['tweet', 'twit']
```

Longest vs Shortest Matches

By default, regular expression libraries return the longest match.

```
In [19]: print re.findall(r'ab+','xyz abbbbbbc def')

         ['abbbbbb']
```

Sometimes, you want the shortest possible match. You get that by putting a ? after a repeat operator like *, +, or ?.

```
In [25]: print re.findall(r'ab+?','xyz abbbbbbc def')

         ['ab']
```

Note that this does not "search for" the shortest match, it is just that when it matches, it picks up the shortest string.

```
In [28]: print re.search(r'ab+?','xyz abbbbbbc abc def').start(0)

         4
```

Grouping

```
In [46]: print re.findall(r'the ([^ ]*)','the cat in the hat')

         ['cat', 'hat']
```

```
In [47]: print re.findall(r'(a|the) ([^ ]*)','a cat in the hat')

         [('a', 'cat'), ('the', 'hat')]
```

```
In [48]: g = re.search(r'(a|the) ([^ ]*)','a cat in the hat')
```

```
In [49]: g.group(0)

Out[49]: 'a cat'
```

```
In [50]: g.group(1)

Out[50]: 'a'
```

```
In [51]: g.group(2)

Out[51]: 'cat'
```

```
In [52]: print g.start(2),g.end(2),g.span(2)

         2 5 (2, 5)
```

```
In [53]: print re.findall(r'(?:a|the) ([^ ]*)','a cat in the hat')

         ['cat', 'hat']
```

```
In [54]: print re.search(r'(the|a) [^ ]+ near \1 [^ ]+','the cat near the cat')
         print re.search(r'(the|a) [^ ]+ near \1 [^ ]+','a cat near a cat')
         print re.search(r'(the|a) [^ ]+ near \1 [^ ]+','the cat near a cat')

         <_sre.SRE_Match object at 0x3b6ff30>
         <_sre.SRE_Match object at 0x3b6ff30>
         None
```

Grouping also takes on special meaning with `split`, alternating between separators and words.

```
In [55]: print re.split(r'([,;]?\s+|\W+$)','The quick, brown fox jumps; over lazy dogs!')

         ['The', ' ', 'quick', ', ', 'brown', ' ', 'fox', ' ', 'jumps', '; ', 'over', '
         ', 'lazy', ' ', 'dogs', '!', '']
```

Named Groups

Grouping can get more complex with naming and conditionals.

In [42]:
```
print re.findall(r'(.)\1','aa bc dd ef')
print re.findall(r'(?P<id>.)(?P=id)','aa bc dd ef')
```

```
['a', 'd']
['a', 'd']
bc
```

Named groups can also be used to refer to parts of patterns.

In [43]:
```
print re.search(r'(?P<id>b.)','aa bc dd ef').group("id")
```

```
bc
```

There are even conditionals based on named groups.

In [47]:
```
q = r'^(<)?[^<>]+(?(1)>|)$'
print re.search(q,'abc')
print re.search(q,'<abc>')
print re.search(q,'<abc')
```

```
<_sre.SRE_Match object at 0x42404e0>
<_sre.SRE_Match object at 0x42404e0>
None
```

Readable Expressions with re.X

Regular expressions can become hard to read very easily.

In [51]:
```
q = r'^(<)?[^<>]+(?(1)>|)$'
```

With the `re.X` flag (or `(?x)`), you can insert whitespace and comments.

In [49]:
```
qx = r"""(?x)

    ^(<)?          # match optional beginning "<"

    [^<>]*         # match any non-bracket character

    (?(1)>|)$      # match a ">" at the end if we did so at the beginning
"""
```

In [50]:
```
print re.search(q,'<abc>')
print re.search(qx,'<abc>')
```

```
<_sre.SRE_Match object at 0x42405d0>
<_sre.SRE_Match object at 0x42405d0>
```

Character Classes

There are a number of common special character classes:

- `\A` - empty string at the beginning of the string
- `\Z` - empty string at end of string
- `\b` - empty string at the beginning of the word
- `\B` - empty string not at the beginning of the word (upper case is often inverse of lower case)
- `\d` - digit (usually `[0-9]`, or digit class in Unicode)
- `\D` - not a digit
- `\s` - white space
- `\S` - not white space
- `\w` - word character
- `\W` - not a word character

```
In [56]: re.findall(r'\w+',"The quick brown fox... jumped over the la$y dogz.")

Out[56]: ['The', 'quick', 'brown', 'fox', 'jumped', 'over', 'the', 'la', 'y', 'dogz']
```

```
In [57]: numbers = re.compile(r'((?:\d+\.\d*|\d*\.\d+)(?:e[+-]\d+)?)',re.I)
         numbers.findall("The fine structure constant is 7.2973525698e-3, and pi is about 3.1

Out[57]: ['7.2973525698e-3', '3.14159']
```

Lookahead and Lookbehind

Sometimes you want to match something "in context" without actually considering the context part of the match. For this, you can use lookahead and lookbehind assertions.

```
In [29]: re.findall(r"[abc](?=z)","ax by cz")

Out[29]: ['c']
```

```
In [30]: re.findall(r"[abc](?!z)","ax by cz")

Out[30]: ['a', 'b']
```

```
In [31]: re.findall(r"(?<=a)[xyz]","ax by cz")

Out[31]: ['x']
```

```
In [32]: re.findall(r"(?<!a)[xyz]","ax by cz")

Out[32]: ['y', 'z']
```

Other Regular Expression Features

Above, we have seen the standard Python regular expression features. Regular expressions differ somewhat between different tools.

Most importantly, quoting differs: special characters like `(`, `)`, and `|` are sometimes special by default, and sometimes need a backslash like `\|` in order to take on their special meaning.

POSIX tools support special POSIX character classes, like `[:upper:]`, `[:digit:]` etc.

Perl supports *recursive regular expressions*; these aren't really "regular expressions" at all anymore, they are more like general purpose parsing. (In Python, there are several parsing modules you can use instead.)

More Powerful Module

There is a more powerful regular expression module in Python, called `regex`.

It handles Unicode better and supports some interesting additional features.

```
In [58]: import regex
```

Recursive Matching

```
In [59]: r = regex.compile(r"^(\w+|\((?1)[+*/-](?1)\))$")
```

```
In [60]: r.match("x")
Out[60]: <_regex.Match at 0x3ce6d98>
```

```
In [61]: r.match("(x+y)")
Out[61]: <_regex.Match at 0x3ce6e00>
```

```
In [62]: r.match("(x*(y+z))")
Out[62]: <_regex.Match at 0x3ce6e68>
```

```
In [63]: r.match("(x*y+z))")
```

Fuzzy Matching

Fuzzy matching allows edit distance information to be taken into account during matching. That is, a group does not need to match precisely.

```
In [147]: regex.findall(r"(?=\w)(quick){e<=1}","the quick brown fox quacks loudly")

Out[147]: ['quick',
           'quack']
```

You can specify the number of insertions, deletions, substitutions, and errors.

Named Lists

Often, it is useful to compile large lists of words into a regular expression (c.f. `fgrep`).

```
In [65]: with open("basic-english.txt") as stream: words = stream.read().split()
         len(words)

Out[65]: 851

In [82]: allwords = regex.compile(r"\b(\L<words>)(?:s|es|ed|ing)?\b(?i)",words=words)

In [83]: allwords.findall("The quick brown fox jumps over the lazy dogs.")

Out[83]: ['The', 'quick', 'brown', 'fox', 'jump', 'over', 'the', 'lazy', 'dog']

In [84]: fuzzywords = regex.compile(r"\b(\L<words>){e<=2}(?:s|es|ed|ing)?\b(?i)",words=words)

In [85]: print fuzzywords.findall("The quock briwn fox jxmps over the lazy dogs.")

         ['The ', 'quock', ' ', 'briwn', ' fox', ' ', 'jxmp', ' over', ' the ', 'lazy', '
         dog', '']

In [87]: fuzzywords = regex.compile(r"\b(?=\w)(\L<words>){e<=2}(?:s|es|ed|ing)?\b(?i)",words=

In [88]: print fuzzywords.findall("The quock briwn fox jxmps over the lazy dogs.")

         ['The ', 'quock', 'briwn', 'fox ', 'jxmp', 'over', 'the ', 'lazy', 'dogs']
```

Better Text and Unicode Support

There is generally better Unicode support in `regex`:

- word characters (`\w` etc.) refer to Unicode by default
- line separators refer to Unicode line separators
- whitespace recognizes Unicode whitespace
- `\m` and `\M` match at the beginning/end of a word respectively
- there are set operators

- POSIX character classes are recognized
- you can access Unicode properties with `\p` and `\P`
- you can match graphemes with `\X`

```
In [99]:  regex.findall(ur'\S+',u'the quick рыжая лиса')

Out[99]:  [u'the',
           u'quick',
           u'\u0440\u044b\u0436\u0430\u044f',
           u'\u043b\u0438\u0441\u0430']
```

```
In [98]:  regex.findall(ur'\w+',u'the quick рыжая лиса')

Out[98]:  [u'the',
           u'quick',
           u'\u0440\u044b\u0436\u0430\u044f',
           u'\u043b\u0438\u0441\u0430']
```

```
In [101]:  regex.findall(ur'\p{Script=Latin}+',u'the quick рыжая лиса')

Out[101]:  [u'the',
           u'quick']
```

```
In [100]:  regex.findall(ur'\p{Script=Cyrillic}+',u'the quick рыжая лиса')

Out[100]:  [u'\u0440\u044b\u0436\u0430\u044f', u'\u043b\u0438\u0441
           \u0430']
```

```
In [104]:  s = u"Käse"
           t = unicodedata.normalize('NFD',s)
           print repr(s)
           print repr(t)

           u'K\xe4se'
           u'Ka\u0308se'
```

By default, `re` doesn't consider non-ASCII characters word characters at all.

```
In [109]:  re.findall(ur"\w",s),re.findall(ur"\w",t)

Out[109]:  ([u'K', u's', u'e'], [u'K', u'a', u's',
           u'e'])
```

With Unicode support, it does, but it doesn't handle decomposed characters.

```
In [111]:  re.findall(ur"\w(?u)",s),re.findall(ur"\w(?u)",t)

Out[111]:  ([u'K', u'\xe4', u's', u'e'], [u'K', u'a', u's',
           u'e'])
```

The `regex` package deals correctly with word characters by default, but still doesn't handle deocmposed characters with either `\w` or `..`

In [110]:
```
regex.findall(ur"\w",s),regex.findall(ur"\w",t)
```

Out[110]:
```
([u'K', u'\xe4', u's', u'e'], [u'K', u'a', u'\u0308', u's',
u'e'])
```

In [107]:
```
regex.findall(ur".",s),regex.findall(ur".",t)
```

Out[107]:
```
([u'K', u'\xe4', u's', u'e'], [u'K', u'a', u'\u0308', u's',
u'e'])
```

However, the grapheme matcher \X recognizes that the decomposed umlaut is, in fact, a single grapheme, even though it consists of several codepoints.

In [108]:
```
regex.findall(ur"\X",s),regex.findall(ur"\X",t)
```

Out[108]:
```
([u'K', u'\xe4', u's', u'e'], [u'K', u'a\u0308', u's',
u'e'])
```

Parsing

Regular expressions are best for fairly simple tasks. For more complex parsing tasks, you may want to use an actual parsing tool, like pyparsing.

In [113]:
```
import pyparsing
```

In [115]:
```
pyparsing.nestedExpr().parseString("(a (b c) d)").asList()
```

Out[115]:
```
[['a', ['b', 'c'],
'd']]
```

In [136]:
```
import string
from pyparsing import oneOf,Literal,Word,Optional,StringEnd
greeting = oneOf("Hi Yo") + Optional(Literal(",")) + Word(string.uppercase,string.lc
```

In [137]:
```
greeting.parseString("Hi, Peter!")
```

Out[137]:
```
(['Hi', ',', 'Peter', '!'],
{})
```

```
In [140]: greeting.parseString("Yo, DogZ.")
```

```
---------------------------------------------------------------------------
ParseException                            Traceback (most recent call last)
<ipython-input-140-1186c0727049> in <module>()
----> 1 greeting.parseString("Yo, DogZ.")

/usr/lib/python2.7/dist-packages/pyparsing.pyc in parseString(self, instring,
parseAll)
   1030                 # catch and re-raise exception from here, clears out
pyparsing internal stack trace
   1031                 exc = sys.exc_info()[1]
-> 1032                 raise exc
   1033         else:
   1034             return tokens

ParseException: Expected end of text (at char 7), (line:1, col:8)
```

```
In [ ]:
```