

# COM6115: Text Processing

*Programming for Text Processing:*

*Regular Expressions*

Mark Hepple

Department of Computer Science  
University of Sheffield

# What are Regular Expressions?

- A regular expression (**regex**) is a *pattern* that describes a *set of strings*
  - ◇ a *matching* process tests if a given string matches the pattern
  - ◇ may also then *modify* the string
    - e.g. by *substituting* a substring, or *splitting* it into substrings
- Regular expressions are a powerful programming tool, used widely in computing / text processing
  - ◇ found (e.g.) in Perl, Tcl/Tk, Java, grep, sed, awk, vi, emacs, lex
  - ◇ *but* note that regex syntax varies
- Many applications — some random examples:
  - ◇ strip the html out of a set of web pages
  - ◇ extract comment blocks from a program (e.g. to build documentation)
  - ◇ check a document for doubled words (“the the”, “here here”)

- Simplest example of a regex is a *literal pattern*
  - ◇ most characters just match against themselves
  - ◇ likewise, most char sequences form a regex to match against identical char sequence in a string
  - ◇ *but* some chars have special behaviour: *metachars*
- For example, string "pen":
  - ◇ as a regex, matches any string containing *substring* pen
    - e.g. "the pen broke"
    - e.g. "what is epenthesis?"

# Simple Patterns — Example: Python

- Python provides extensive regex facilities
  - ◇ not in basic language — must import module "re"
  - ◇ can do regex matching using module functions 'directly'

- Example:

```
import sys, re
with open(sys.argv[1], 'r') as infs:
    for line in infs:
        if re.search('pen', line):
            print(line, end='')
```

- ◇ **search** scans for *first* substring matching regex *anywhere* in string
- ◇ if finds match, returns a *match object*, else returns None (=False)

## Simple Patterns — Example: Python (ctd)

- When a regex is to be used many times, is better (i.e. faster) to *compile* a regex *object*

- Example:

```
import sys, re
penRE = re.compile('pen')
with open(sys.argv[1], 'r') as infs:
    for line in infs:
        if penRE.search(line):
            print(line, end='')
```

- Assigning object to a *well-named variable* gives more-readable code  
e.g. having regexes for 'word', 'URL', etc

# Alternatives in Regexes and Groupings

- To specify that one of several options are permitted in a match, separate them by a *vertical bar* (or 'pipe')

◇ *Example:* regex "car|bike|train" matches any of:

carnation  
motorbike  
detraining

- Can *group* parts of a pattern, using *parentheses*

◇ *Example:* regex "(e|i)nquir(e|y|ing)" matches any of:

enquiry  
inquiring  
enquire

- *Quantifiers*: specify want *some number* of (sub)pattern occurrences
- Following quantifiers based on *Kleene* notation:

*	zero or more
+	one or more
?	zero or one occurrences (i.e. optional)

- ◇ apply to *immediately preceding* item (or group) in pattern
- ◇ e.g. does regex "ab\*d?e" match ... (✓ = yes; ✗ = no)
  - abde (✓), bde (✗), abd (✗), aeeee (✓), abbbbbbbbbde (✓)
- ◇ e.g. does regex "c(ab)\*(de) +" match ...
  - ghcabdemn (✓), ghcabbdemn (✗), ghcababdemn (✓)
- There's an alternative notation for quantifiers, using braces ({,})
  - ◇ allows *count range* for repetitions to be specified (e.g. "3-12")
  - ◇ see *extended presentation* slides

# Character classes

- Use *square brackets* (`[,]`) to indicate a character class
  - ◇ allows *alternatives* for match to a *single char*  
e.g. regex `"c[ad]r"` matches `cdr` and `car` but not `cadr` and
- Can specify char *ranges* using a hyphen, e.g.
  - ◇ `[A-Z]` upper case roman alphabet
  - ◇ `[a-f]` lower case letters a..f
  - ◇ `[A-Za-z]` upper and lower case letters
  - ◇ `[0-9]` digits 0..9
- Some common char classes have *predefined* names:

<code>.</code>	matches <i>any</i> char
<code>\d</code>	abbreviates <code>[0-9]</code>
<code>\w</code>	abbreviates <code>[A-Za-z0-9_]</code>
<code>\s</code>	abbreviates <code>[ \f\t\n\r]</code> (i.e. <i>whitespace</i> chars)



# Negated Character Classes & Anchors

- To *negate* a char class, put the “carat” sign `^` at the start
  - ◇ matches anything *except* chars indicated. e.g. `[^0-9]`
- Some negated char classes are *predefined*  
e.g. `\D` (not 0-9), `\S` (not whitespace), `\W` (not `\w`)
- *Anchors* tie matching to appear at certain positions:
  - ◇ `^` matches the *beginning* of the string
  - ◇ `$` matches the *end* of the string
  - ◇ `\b` matches at word *boundary* (between `\w` and `\W`)  
(*but* see *extended presentation* slide on *raw strings* before using `\b`)

e.g. `"^author"` — matches strings *beginning with author*

e.g. `">>$"` — matches strings *ending with »*

# Extracting matched parts

- Use brackets (*groups*) in regex to identify portions for return
  - ◊ identified *numerically* – count '('s in from left, starting with 1e.g. in '(([a-z]+)(ed|ing))', gp 2 is '([a-z]+)', gp 3 is '(ed|ing)'
- A successful regex match returns a *match object*:
  - ◊ stores info of matching substrings and their spans
  - ◊ access using match object's methods: *group*, *groups*, *span*

```
>>> sent = "I have baked a cake!"
>>> m = re.search(' (([a-z]+)(ed|ing)) ', sent)
>>> m
<_sre.SRE_Match object at 0x1081b5030>
>>> m.group(1)    # returns substring for group 1
'baked'
>>> m.span(1)     # returns start/end indices for group 1
(7, 12)
>>> m.group(3)    # substring for group 3
'ed'
>>> m.span(3)     # start/end indices for group 3
(10, 12)
```

# Finding Multiple Regex Matches

- `findall` method returns a *list of matches* for regex, e.g.:

```
>>> s = 'I like fish, chips and peas!'
>>> word = re.compile('[A-Za-z]+')
>>> word.findall(s)
['I', 'like', 'fish', 'chips', 'and', 'peas']
>>>
```

- ◇ above regex *has no groups*
  - in this case, list of matching strings returned
- ◇ in case where regex *has groups*, instead returns a list of n-tuples of group matches (for groups 1+)

# Python String Methods

- Worth knowing that various methods of the Python `string` type are also useful for text manipulation
  - ◇ are *simpler/more efficient* than regexes
  - ◇ method names provide a *clear 'semantics'* to code
- Some examples:
  - ◇ *string testing* methods such as:
    - `isupper()`, `islower()`, `isalpha()`, `isdigit()`, `isalnum()`
  - ◇ upper/lower *case conversion*: `upper()`, `lower()`, `capitalize()`
  - ◇ *splitting* on a fixed string (not regex): `split()`
    - default 'split string' is space, but can specify alternative as arg

```
>>> "hello, world!".upper()
'HELLO, WORLD!'
>>> "hello, world!".split()
['hello,', 'world!']
```

- See the *extended presentation* slides on Regular Expressions, for more info on the above, plus additional topics, including:
  - ◇ Changing the *properties of matching*
    - e.g. *greedy* vs. *non-greedy* matching
    - e.g. *case-insensitive* matching
  - ◇ Regex-controlled *string substitution*
  - ◇ Regex-controlled *string splitting*