



IFB104 — Building IT Systems

Topic 5 — How to Find Patterns

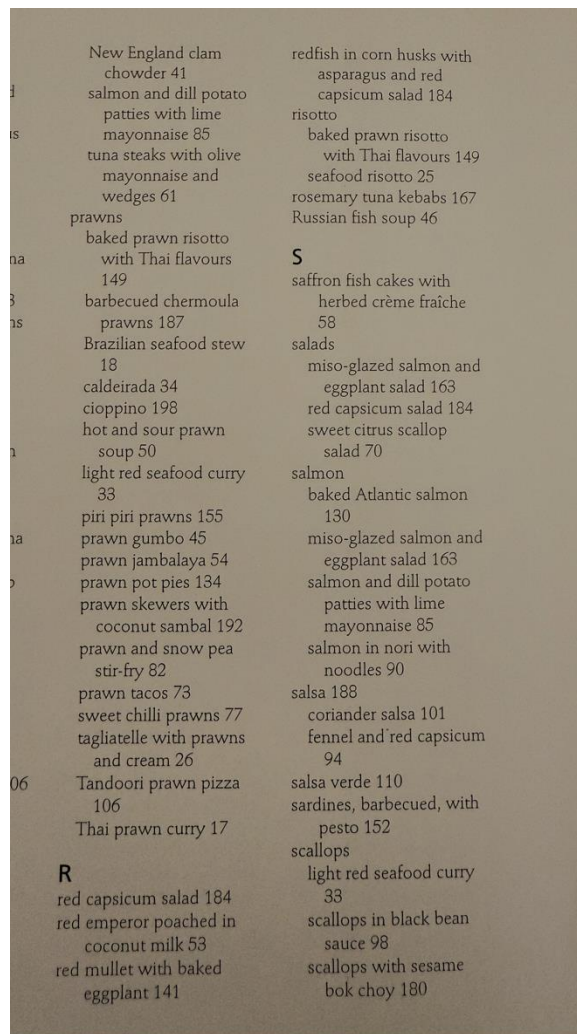
School of Computer Science
Semester 1, 2025

Assessment task 1B

- Keep working on Assessment Task 1B
 - Deadline is Friday of Week 6
 - You're welcome to change your Task 1A code when working on Task 1B
 - Topics for this assignment were covered in Weeks 1 to 5
 - We are still working on Gradescope for immediate feedback, the submissions will open soon.
- There are many ways to make the code work
 - Whether your code works or not is only one of the criteria
 - You are also marked on how you approached the functions
 - Each should use at least a loop or a regular expression
 - Keep an eye on discussion board (and FAQ). We will likely update with clarifications later this week.

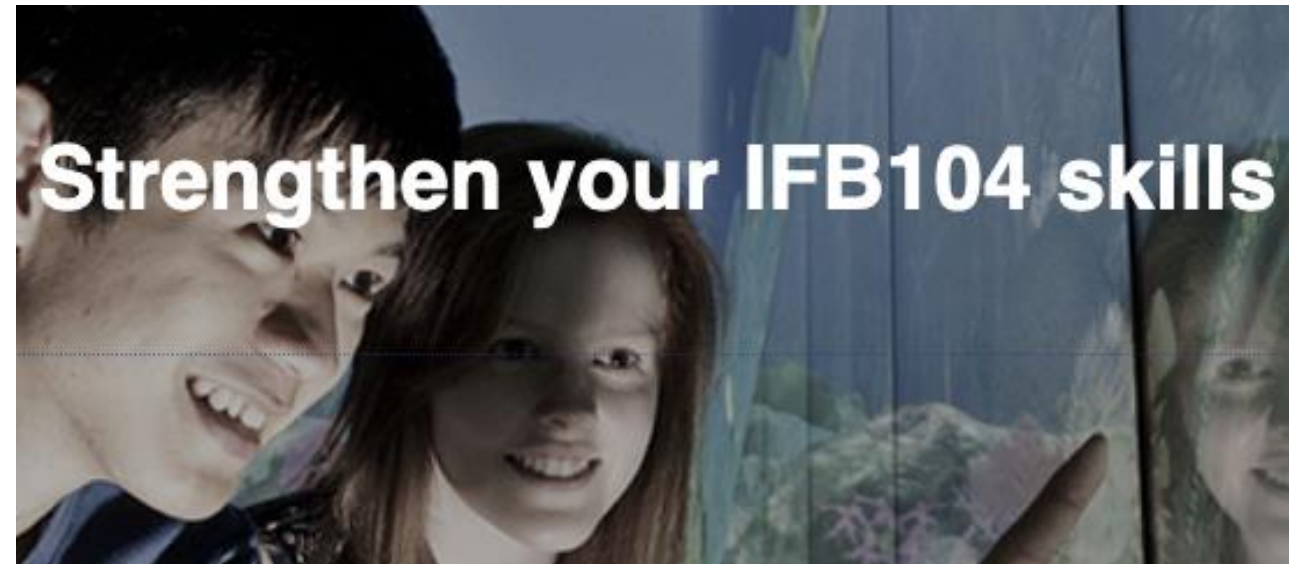
1B: Make Index

- Returns a list of strings and lists of numbers
 - This is a compressed version of the input string. Each word appears only once.
 - For a very long booklist, it would speed up searching for words.
 - This replicates the structure of dictionaries, which we don't cover in IFB104



Need help?

- There is one more *Student Success Group* special workshop for IFB104 students before the deadline
 - Wednesday afternoon next week, on-campus
- Go to the SSG site for full details and to register:
<https://unihub.qut.edu.au/students/events/search?Text=ifb104>
- See Canvas *Unit Overview* | *Getting Help* for links to all the SSG IFB104 help services



Aims of this week's lecture

- To introduce principles of testing and debugging
- To introduce the basics of finding patterns in large amounts of text, including web documents
 - Finding fixed patterns in text using `find`
 - Finding general patterns with regular expressions using `findall`
 - Editing text automatically



Part A — Proving Our Functions Correct by Testing

Modules

- In Python a *module* is any file containing one or more function definitions
- Given a file `m.py` containing function definitions `f`, `g`, `h`, etc we can import all or part of module `m` into our program as desired:

`from m import f, h` # imports just specific functions `f` and `h`

`from m import *` # imports all functions from module `m`

The main program

- If we try to run a program (module) which contains several functions, where do we start?
- Most programming languages introduce a special “main program” function for this purpose
- However, a Python script will treat any code *not* within a function definition as the main program
- In Python we can also test to see whether or not the current module is being run as a main program or if it is being imported into another program

```
if __name__ == '__main__':  
    # Execute this code only if  
    # we are "running" THIS file
```


Testing your functions

- Now that we know how to break programs up into separate functions, how do we convince ourselves they work correctly?
- There are various ways to prove programs correct, but by far the most common is rigorous testing
 - Unit testing — testing individual functions separately
 - Integration testing — testing how well several functions work together
- Good programming practice is to document all the tests you did on your code
 - This allows you to repeat the tests if the code is changed
 - The tests serve as a requirements specification for your code



Documenting our unit tests

- In Python the convention is to document tests in a *triple-quoted* 'doctest' string `"""..."""`
- They are written in exactly the same format they would appear in the Python interpreter (shell)
 - This allows us to directly copy tests from the shell into the program and vice versa
- Assume we want to test the following function

```
# Square a given number
def square(base):
    return base ** 2
```

- To do so we define a suite of tests that show how the function is intended to work
 - A well-designed test suite usually aims to include both 'normal' cases as well as 'boundary' cases

Example of some unit test documentation

```
"""
```

```
Test 1. Zero case:
```

```
>>> square(0)
```

```
0
```

```
Test 2. Positive case:
```

```
>>> square(1)
```

```
1
```

```
Test 3. Positive case:
```

```
>>> square(10)
```

```
100
```

```
Test 4. Negative case:
```

```
>>> square(-5)
```

```
25
```

```
"""
```

Triple-quotes tell us that this is a doctest string

We should explain the purpose of each test

A function call just as it would appear in the shell, including the function's name and argument(s)

The expected result when the function is called with this argument

Edge Cases

- Edge cases are unusual situations that occur at the extreme ends of input ranges or under special conditions.
 - **Empty Inputs:** Test with empty lists, strings, or None values.
 - **Boundary Values:** Test just below, at, and just above the limits (e.g., 0, 1, -1).
 - **Large Inputs:** Test with very large numbers or long strings.
 - **Invalid Inputs:** Test with out-of-range values.
 - **Special Characters:** Test with special characters, whitespace, or escape sequences.

```
"""
```

```
Test 1. Zero case:
```

```
>>> square(0)
```

```
0
```

```
Test 2. Positive case:
```

```
>>> square(1)
```

```
1
```

```
Test 3. Positive case:
```

```
>>> square(10)
```

```
100
```

```
Test 4. Negative case:
```

```
>>> square(-5)
```

```
25
```

```
"""
```

Automated testing in Python

- If we need to repeatedly test our program while it is being developed or improved, it is impractical to manually run lots of tests each time the code is changed
- Most modern programming languages provide a way of automatically running a whole suite of tests all at once
- In Python all the tests in a 'doctest' string can be run automatically by calling the `testmod` function

```
from doctest import testmod  
  
testmod(...)
```

Example of automated testing

- Using the following code as the main program in a module containing some function definitions and a doctest string will cause the tests to be performed when the module is run
 - However the tests will *not* be performed if the module is imported into another program

```
# If this file is being run as the
# main program, automatically execute
# any tests it contains

if __name__ == '__main__':
    from doctest import testmod
    testmod(verbose = True)
```

24/03/2025

TEQSA Provider ID PRV12079 Australian University | CRICOS No. 00213J

Demonstration file: distance_doctest.py

Part B —Debugging Principles

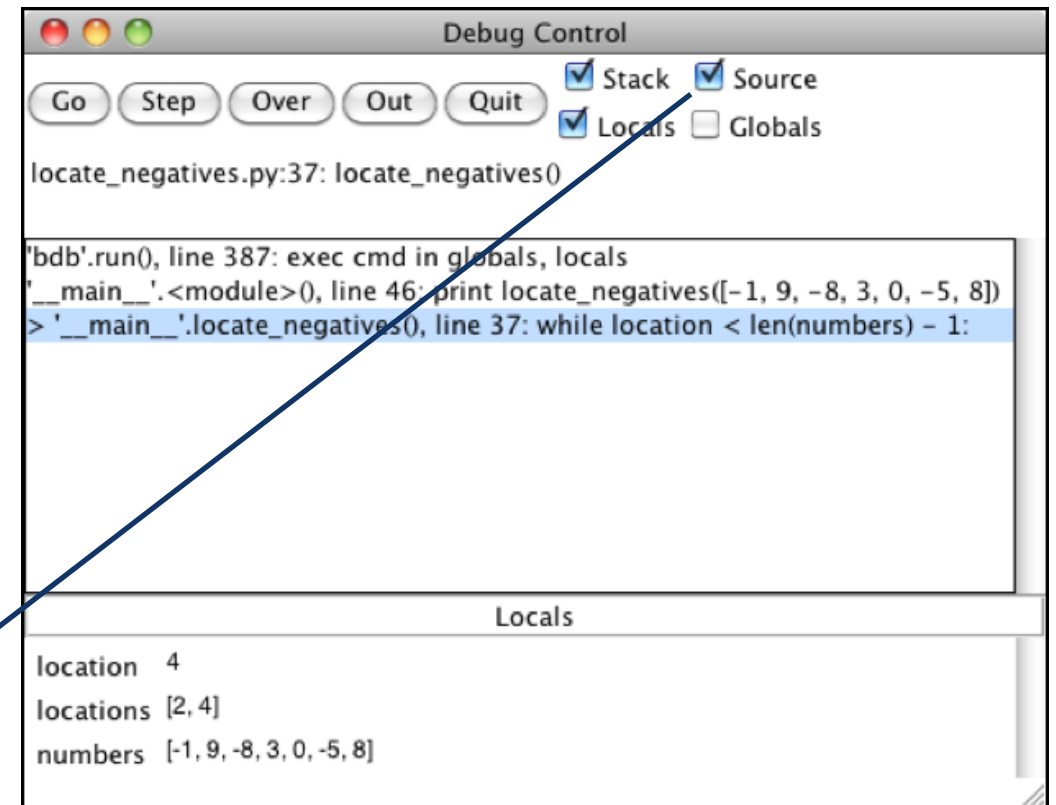
Debugging program code

- Our programs are now getting complicated
 - When things go wrong it can be hard to understand why
- Debugging is the process of removing semantic errors from an IT system
 - Programs that “crash”
 - Programs that run but don’t do what we want
- Debugging involves two steps:
 1. Identifying the problem (hard)
 2. Fixing it (easier)
- The first of these can be tackled in three different ways:
 1. Manual execution of the code by creating a trace table showing the value of each variable after each statement is executed
 2. Inserting calls to the `print` function to display variable values at key points in the code
 3. Using a debugging tool to display all variable values either at each step or at selected ‘breakpoints’

Debugging in IDLE

- In simple cases just adding a few calls to `print` to your code can be enough to help you diagnose a bug
- In more complex situations an Interactive Development Environment like IDLE provides a debugger that lets you watch code execute one statement at a time

Make sure this box is ticked so that statements in the source code are highlighted during execution



The most important debugging aid of all

- This is the best tool to use for debugging programs:

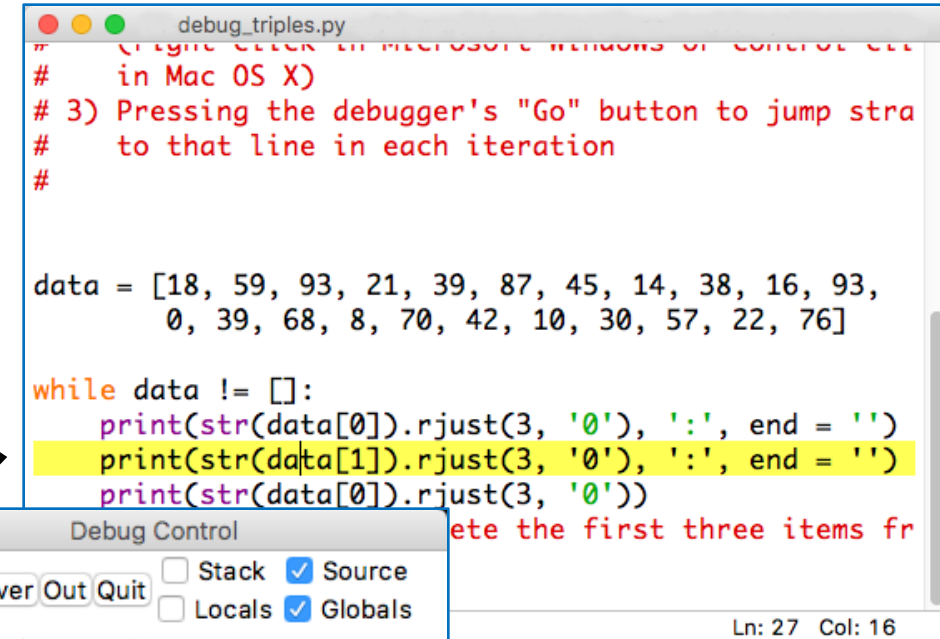


- The basic rule for engaging it is as follows:



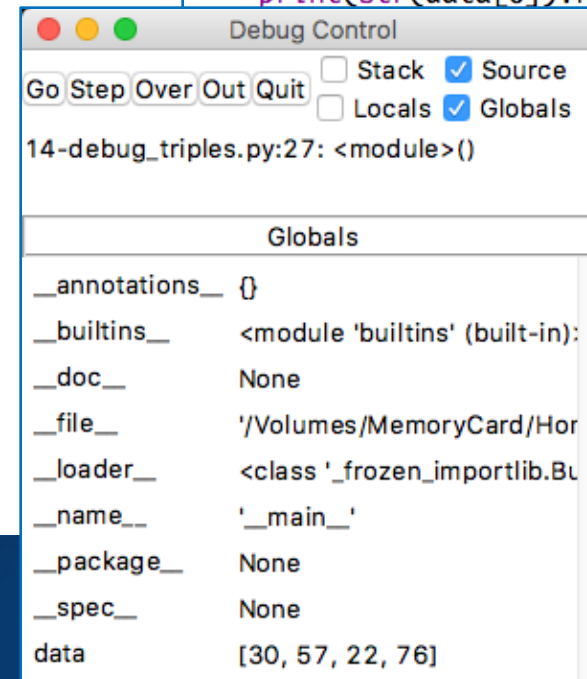
The challenge of debugging iterative programs

- When programs do repeated actions *many* times, 'single stepping' through the code is too laborious and time-consuming
- Instead we can set *breakpoints* in the program to tell execution to stop on a certain line
 - Right-click on the line in Microsoft Windows or control-click in macOS
 - The debugger's *Go* button will run the code until the breakpoint is reached



```
debug_triples.py
# (Right-click in Microsoft Windows or control-click
# in Mac OS X)
# 3) Pressing the debugger's "Go" button to jump straight
# to that line in each iteration
#
data = [18, 59, 93, 21, 39, 87, 45, 14, 38, 16, 93,
        0, 39, 68, 8, 70, 42, 10, 30, 57, 22, 76]

while data != []:
    print(str(data[0]).rjust(3, '0'), ': ', end = '')
    print(str(data[1]).rjust(3, '0'), ': ', end = '')
    print(str(data[0]).rjust(3, '0'))
    # delete the first three items from data
```



Part C — Finding Fixed Text Patterns in Character Strings

Finding substrings

- Python's built-in `find` method for character strings provides a simple way of finding fixed substrings in some text
`text.find(substring)`
- It returns the position of the *first* occurrence of the substring in the text
 - It returns `-1` if the substring doesn't occur in the text at all
 - The `rfind` method does the same thing but searches from the right so it finds the *last* occurrence

```
>>> tongue_twister = \
    "The sixth sick sheik's sixth sheep's sick"
>>> # Where does the word "sick" first appear?
>>> tongue_twister.find("sick")
10
>>> # Where does it appear last?
>>> tongue_twister.rfind("sick")
37
>>> # Where does the word "fifth" appear?
>>> tongue_twister.rfind("fifth")
-1
```

Finding multiple occurrences of substrings

- You can also search from a particular position

`text.find(substring, starting_pos)`

- Combining this capability with loops allows *all* occurrences of a substring to be found
- In general, however, we need a more powerful way of expressing patterns if we want to find multiple occurrences of complex patterns ...

```
>>> tongue_twister = \
    "The sixth sick sheik's sixth sheep's sick"
>>> # Print all the positions where string "si" appears
>>> position = tongue_twister.find("si")
>>> while position != -1:
    print("'si' appears at position", position)
    position = tongue_twister.find("si", position + 1)
```

```
'si' appears at position 4
'si' appears at position 10
'si' appears at position 23
'si' appears at position 37
```


Part D — Finding General Patterns in Text Using Regular Expressions

Motivatic

WHENEVER I LEARN A NEW SKILL I CONCOCT ELABORATE FANTASY SCENARIOS WHERE IT LETS ME SAVE THE DAY.

OH NO! THE KILLER MUST HAVE FOLLOWED HER ON VACATION!



BUT TO FIND THEM WE'D HAVE TO SEARCH THROUGH 200 MB OF EMAILS LOOKING FOR SOMETHING FORMATTED LIKE AN ADDRESS!



IT'S HOPELESS!

EVERYBODY STAND BACK.

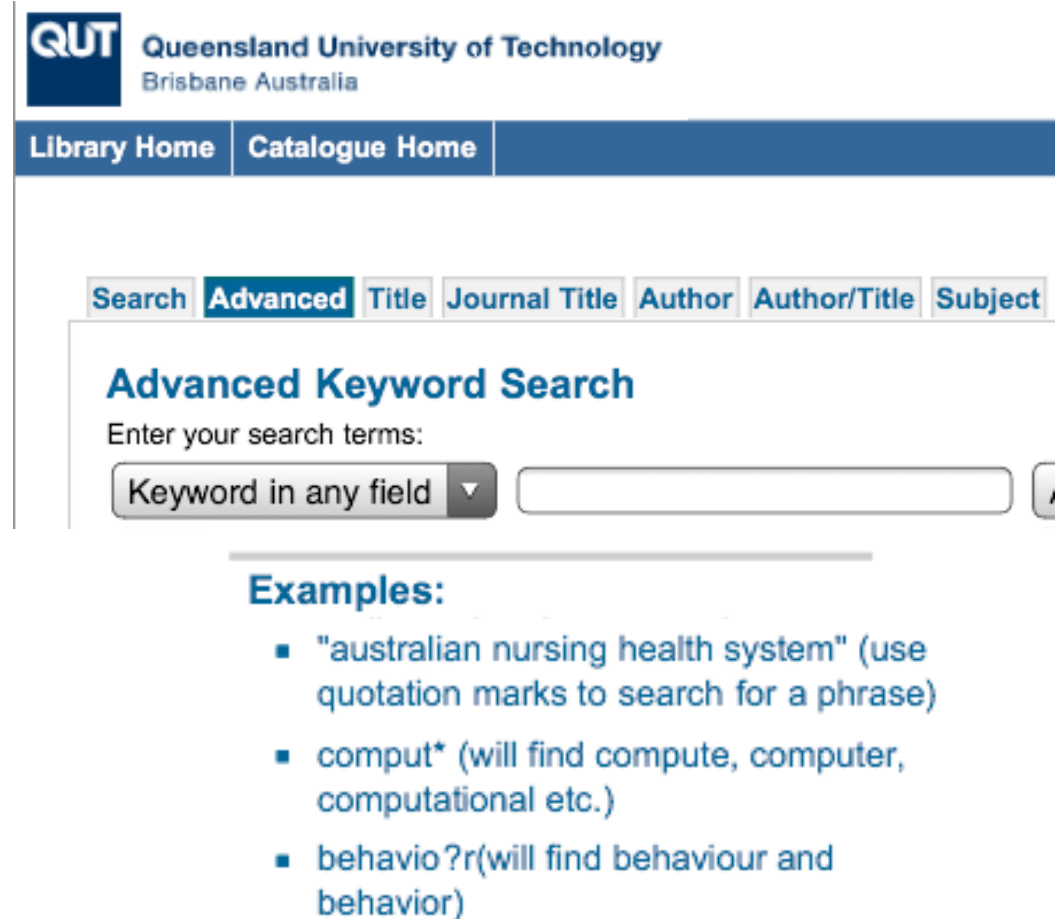


I KNOW REGULAR EXPRESSIONS.



A first taste of regular expressions

- *Regular expressions* are a notation for describing patterns consisting of sequences of symbols (characters)
- You have *already* been using a simplified form of regular expressions in your search terms



The screenshot displays the QUT library website. At the top, the QUT logo and name 'Queensland University of Technology Brisbane Australia' are visible. Below this is a navigation bar with 'Library Home' and 'Catalogue Home' links. The main search area features a row of tabs: 'Search', 'Advanced' (which is selected and highlighted in blue), 'Title', 'Journal Title', 'Author', 'Author/Title', and 'Subject'. Under the 'Advanced' tab, the heading 'Advanced Keyword Search' is shown. Below the heading, it says 'Enter your search terms:' followed by a search input field. To the left of the input field is a dropdown menu currently set to 'Keyword in any field'. Below the search section, there is a section titled 'Examples:' with three bullet points: '■ "australian nursing health system" (use quotation marks to search for a phrase)', '■ comput* (will find compute, computer, computational etc.)', and '■ behavio?r(will find behaviour and behavior)'.

QUT Queensland University of Technology
Brisbane Australia

Library Home Catalogue Home

Search **Advanced** Title Journal Title Author Author/Title Subject

Advanced Keyword Search

Enter your search terms:

Keyword in any field

Examples:

- "australian nursing health system" (use quotation marks to search for a phrase)
- comput* (will find compute, computer, computational etc.)
- behavio?r(will find behaviour and behavior)

So just what are regular expressions?

- Regular expressions are simply a more sophisticated version of these search terms
- There is no standardised syntax for regular expressions, but by convention certain operators are commonly used, including choice '[...]', repetition '*', grouping '(...)', etc



Some regular expression primitives

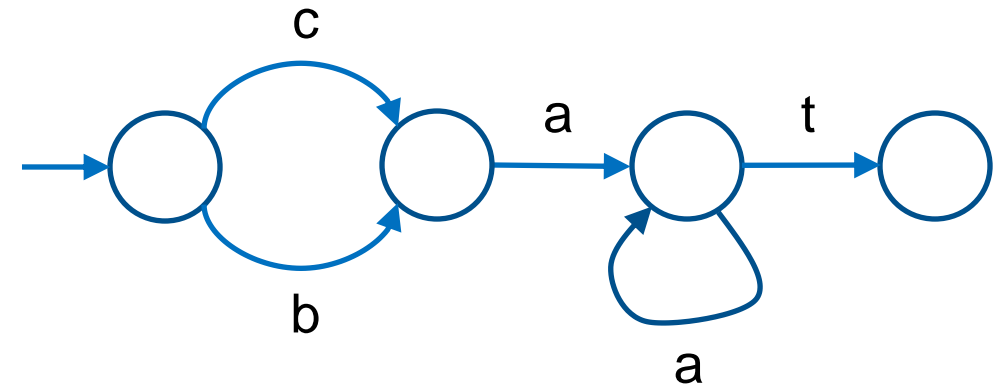
a	The letter “a”
ab	The letter “a” followed by the letter “b”
[abc]	Any one of the letters “a”, “b” or “c”
.	Any single character (except newlines, unless in “dotall” mode)
[b-m]	Any one of the letters from “b” to “m”, inclusive
[^xy]	Any one character <i>except</i> the letters “x” or “y”
^	The beginning of the string (or of a line in “multi-line” mode)
\$	The end of the string (or of a line in “multi-line” mode)
\b	The beginning or ending of a word
\s	Any whitespace character (space, tab, newline, etc)
\n	A newline character (in a multi-line string)

Some regular expression operators

P^*	Pattern P repeated zero or more times
P^+	Pattern P repeated one or more times
$P^?$	Pattern P zero times or once (i.e., P is optional)
$P\{n\}$	Pattern P repeated exactly n times
$P \mid Q$	Pattern P or pattern Q
$(...)$	Several patterns treated as one group (see later)

Regular expressions as language generators

- One way of thinking about regular expressions is as language *generators*
- Example — Regular expression `[cb]a+t` defines the following strings:
`cat, bat, caat, baat, caaat, baaat, ...`
- Regular expressions are powerful because they allow us to describe very large sets of symbol sequences concisely
 - The short regular expression above describes an *infinite* number of strings!
 - Regular expressions both define and can be recognised by finite state machines, and are thus a core part of computer science *theory*



Practicing with regular expressions

- There are many good tools online to help you develop regular expressions
- The following slides contain a few examples that we've tried
 - The first two have been tested against the workshop exercises and are recommended



pythex (<https://pythex.org/>)

Easy access
to options

Very simple
user interface

Groups
created
using (...)
are shown
separately

pythex

[Link to this regex](#)

Your regular expression:

IGNORECASE

MULTILINE

DOTALL

VERBOSE

Your test string:

The Adventures of Superman

Faster than a speeding bullet!
More powerful than a locomotive!
Able to leap tall buildings in a single bound!

"Look, up in the sky!"
"It's a bird!"
"It's a plane!"
"It's Superman!!!"

Yes, it's Superman, strange visitor from another planet who
came to Earth with powers and abilities far beyond those of

Match result:

The Adventures of Superman

Faster than a speeding bullet!
More powerful than a locomotive!
Able to leap tall buildings in a single bound!

"Look, up in the sky!"
"It's a bird!"
"It's a plane!"
"It's Superman!!!"

Yes, it's Superman, strange visitor from another planet who

Match captures:

No groups.
No groups.
No groups.
No groups.
No groups.
No groups.

regular expressions 101

(<https://regex101.com/>)

Choose
Python

regular expressions 101

@regex101 \$ donate sponsor contact bug reports & feedback wiki whats new?

</>

SAVE & SHARE

Save Regex %+\$

FLAVOR

</> PCRE2 (PHP >=7.3)

</> PCRE (PHP <7.3)

</> ECMAScript (JavaScri...

</> Python ✓

</> Golang

</> Java 8

FUNCTION

> Match ✓

Substitution

List

Unit Tests

TOOLS

Code Generator

SPONSOR

Layer0

Jamstack at Scale

REGULAR EXPRESSION

19 matches (557 steps, 17.0ms)

`[A-Z][a-z']+` gm

TEST STRING

The Adventures of Superman
Faster than a speeding bullet!
More powerful than a locomotive!
Able to leap tall buildings in a single bound!
"Look, up in the sky!"
"It's a bird!"
"It's a plane!"
"It's Superman!!!"
Yes, it's Superman, strange visitor from another planet who came to Earth with powers and abilities far beyond those of mortal men... Superman, who can change the course of mighty rivers, bend steel in his bare hands, and who--disguised as Clark Kent, mild-mannered reporter for a great Metropolitan newspaper-- fights a never-ending battle for truth, justice and the American way.

EXPLANATION

`[A-Z][a-z']+` gm
Match a single character present in the list below
w
`[A-Z]`
`A-Z` matches a single character in the range between `A` (index 65) and `Z` (index 90) (case sensitive)

MATCH INFORMATION

Match 1

0-3

The

Match 2

4-14

Adventures

Match 3

18-26

Superman

QUICK REFERENCE

Search reference

All Tokens

★ Common Tok... ✓

General Tokens

⌚ Anchors

A single ch... [abc]

A charact... [^abc]

A character... [a-z]

A charact... [^a-z]

A chara... [a-zA-Z]

Explains
regex
textually

TEQSA Provider ID PRV12079 Australian University | CRICOS no. 00017W

QUT

CyrilEx (<https://extendsclass.com/regex-tester.html>)

ExtendsClass Free Online Toolbox for developers **CyrilEx** Regex Tester

Buy me a coffee About My Account

RegEx Engine Save & Share 19 matches

Quick Start

- ☐ JavaScript
- ☒ **Python (3.4)**
- ☐ Ruby (2.1)
- ☐ Java (JDK 14)
- ☐ PHP (7)
- ☐ MySQL 8.0 (beta)

Regular expression to test

/[A-Z][a-z]+/g

String to test

Generate a string from RegEx (Beta)

```
1 The Adventures of Superman
2
3 Faster than a speeding bullet!
4 More powerful than a locomotive!
5 Able to leap tall buildings in a single bound!
6
7 "Look, up in the sky!"
```

Explanation

RegExp: /[A-Z][a-z]+/g

One of: [A-Z] One of: [a-z]

1 or more times

Choose
Python

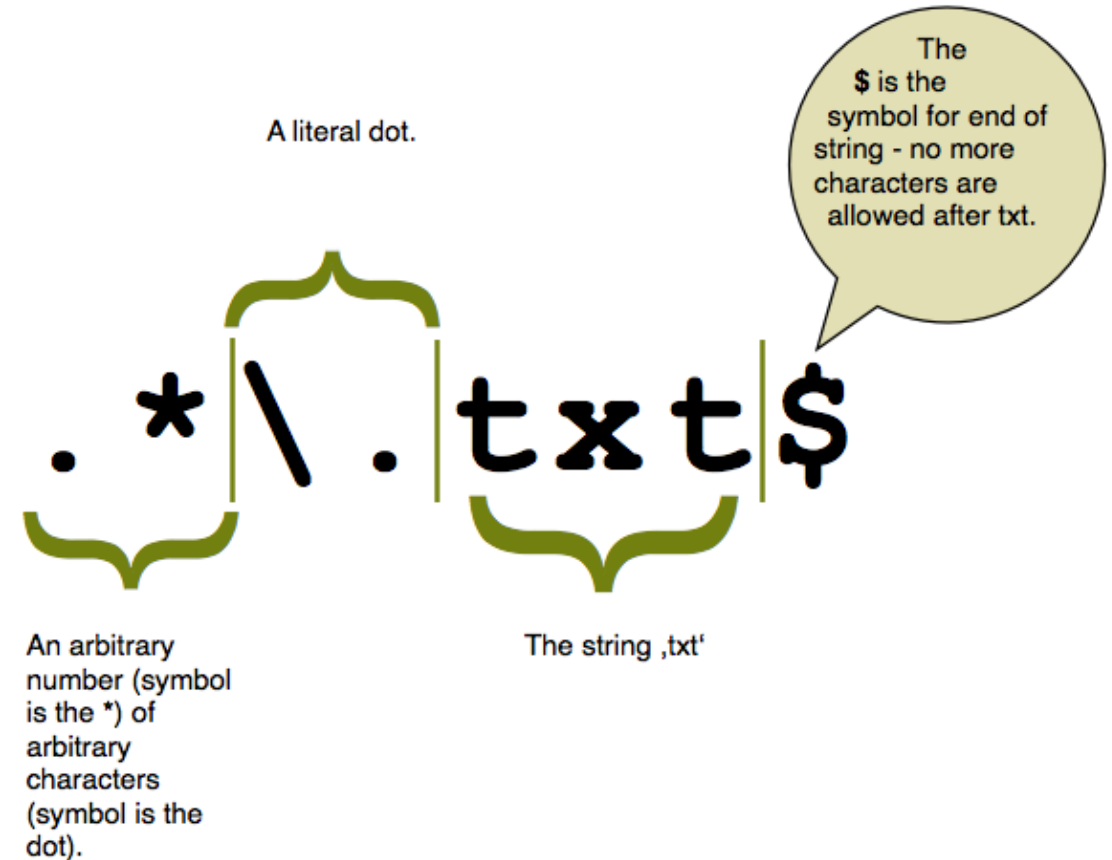
Doesn't display
groups here
when using (...)
brackets

Explains
regex
visually

Key programming concept: How to read regular expressions



- At first glance regular expressions look complicated
- The secret to understanding them is to read them from left to right and break them down into their individual parts
- The pattern on the right matches filenames with a '.txt' extension appearing at the end of a line of text (in multi-line mode)



Key programming concept: How to write regular expressions



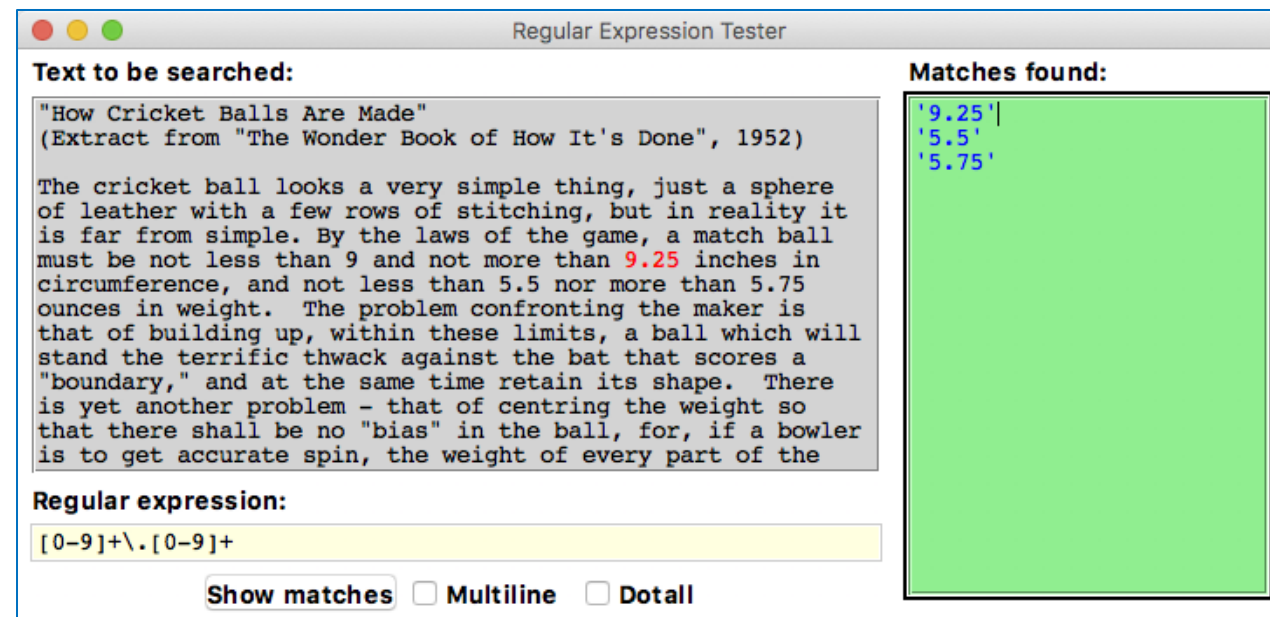
- Perfecting a regular expression can be *very frustrating*fun:
 - An expression which is too “generous” will match lots of unwanted strings
 - More commonly, an expression which is too specific returns no matches at all (and gives you no idea how to fix it!)
- Again, however, the secret is to break the pattern down into its component parts and think about them one-by-one

One possible regex development strategy:

1. Identify a unique pattern *A* that *always* appears at the beginning of the thing you’re looking for (either at its start or immediately before it)
2. Identify a unique pattern *Z* that *always* appears at the end of the thing you’re looking for (either at its end or immediately after it)
3. Characterise the “body” of the thing you’re searching for (which must *never* include occurrences of patterns *A* or *Z*) as pattern *P*
4. If you *don’t* want to include endpoint markers *A* and *Z* in the results then your regex is *A(P)Z*
5. If you *do* want to include the endpoint markers then the overall regex is *APZ*

Using special characters in patterns

- We have seen that many characters, including `‘.^$*+?{}[]\|()’`, have a special meaning in regular expressions
- To use these meta-characters as ordinary characters in patterns, precede them with a backslash
 - For example, `‘$’` matches the end of the string or line, but `‘\ $’` matches a dollar sign
 - Similarly, `‘.’` matches any character, but `‘\ .’` matches a full stop only
 - However, the meta-characters don’t have their special meaning when they appear in a set `‘[...]’`



Finding patterns in strings using Python

- The most common use of regular expressions is to help us find occurrences of a particular pattern in a long (possibly multi-line) string
 - The string may contain plain english or any other language such as the HTML source code for a web page
- Python's `re` module supports finding patterns via its `findall` function

```
from re import findall
```

```
>>> from re import findall
>>> numbers = '''
The decimal number 185 is the same as
hexadecimal number B9 and octal number 271.
'''
>>> findall('m[aeiou]', numbers)
['ma', 'me', 'ma']
>>> findall('[0-9A-F]+', numbers)
['185', 'B9', '271']
>>> findall('[a-z]*x[a-z]*', numbers)
['hexadecimal']
```

Matching greedily

- Some of the regular expression operators are *greedy*, meaning they will match the longest possible pattern when there is a choice
- Example:
 - Regular expression `'t.*o'` matches any string beginning with `'t'` and ending with `'o'`
 - The following sentence
`We ordered tea for two`
contains three such strings, `'tea fo'`, `'two'` and `'tea for two'`
 - However, a greedy match will return only the *longest* alternative, `'tea for two'`

```
>>> from re import findall
>>> tongue_twister = \
    "The sixth sick sheik's sixth sheep's sick"
>>> findall("si[a-z]*th", tongue_twister)
['sixth', 'sixth']
>>> findall("si.*th", tongue_twister)
["sixth sick sheik's sixth"]
```

Regular expressions vs Python strings

- We've seen that certain characters have a special meaning in regular expressions, e.g., `\[` means match left square bracket `[` literally, rather than start a set
- Confusingly, Python strings *also* use backslashes to give some characters a special meaning, e.g., `\t` is a tab character in Python strings
- When representing a regular expression as a Python string we can avoid ambiguous uses of special characters by preceding the string in Python with an `r` so that backslashes are not interpreted as special
 - The Python interpreter does not attempt to process special characters in “raw” strings

```
>>> # The backslashes in this string are not special
>>> file_path = 'C:\Program Files\Java\jdk1.6.0_22'
>>> print(file_path)
C:\Program Files\Java\jdk1.6.0_22
>>>
>>> # But in this case '\t' is interpreted as a tab
>>> file_path = 'C:\Program Files\Java\tkinter95'
>>> print(file_path)
C:\Program Files\Java      tkinter95
>>>
>>> # To solve this we make it a 'raw' string
>>> file_path = r'C:\Program Files\Java\tkinter95'
>>> print(file_path)
C:\Program Files\Java\tkinter95
```

The findall function and groups

- Python's `findall` function has a special behaviour with respect to groups, `'(...)'`:
 - If the pattern doesn't contain a group then the entire matched text is returned
 - If the pattern contains one group then only the text within the group is returned
 - If the pattern contains multiple groups then each of the individual groups is returned
- If you *don't* want this behaviour, e.g., you just want the whole match returned, then you need to use 'non-capturing' brackets for grouping, written `'(?:...)'`

```
>>> batman_family = '''
Some members of the "Batman Family" are
Batgirl, Ace the Bat-Hound and Bat-Mite.
'''

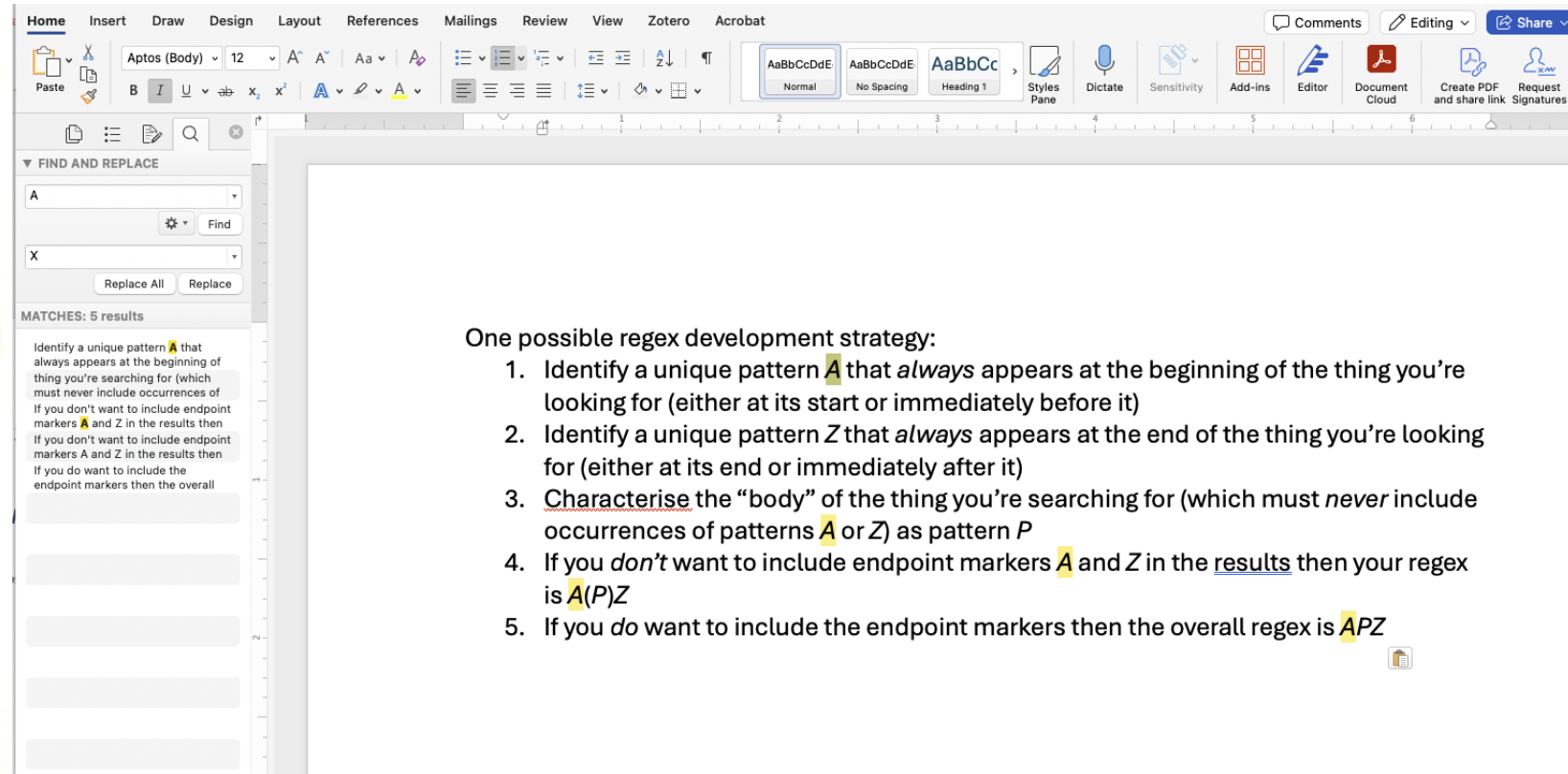
>>> # Return whole match
>>> findall('Bat-[A-Za-z]+', batman_family)
['Batman', 'Batgirl', 'Bat-Hound', 'Bat-Mite']
>>> # Return second part of match
>>> findall('Bat-?([A-Za-z]+)', batman_family)
['man', 'girl', 'Hound', 'Mite']
>>> # Return two groups per match
>>> findall('(Bat-?)([A-Za-z]+)', batman_family)
[('Bat', 'man'), ('Bat', 'girl'), ('Bat-', 'Hound'), ('Bat-', 'Mite')]
>>> # Ignore second group
>>> findall('(Bat-?)(?:[A-Za-z]+)', batman_family)
['Bat', 'Bat', 'Bat-', 'Bat-']
```

Searching for a pattern in a text file

- We often want to search for patterns in text files (including HTML documents)
- It's easy to do this by reading in each line and searching through it
- However, this will miss occurrences that span two or more lines
- A simple solution is to read the *whole* file into a single string
 - But we then need to allow for the possibility of newline characters (`\n`) appearing in the string when defining the pattern

```
>>> text_file = open('documents/Superman.txt')
>>> file_contents = text_file.read()
>>> findall('[a-z]+\n[a-z]+', file_contents)
['who\ncame', 'of\nmortal', 'mighty\nrivers',
'disguised\nas', 'for\ntruth']
```

Modifying Text Using Regular Expressions



This Photo by Unknown Author is licensed under [CC BY-SA-NC](#)
 TEQSA Provider ID PRV12079 Australian University | CRICOS No. 00213J

Replacing patterns found in strings

- Having found an occurrence of a pattern in a string we often want to do something to it
- Python's `sub` function allows each instance of a pattern to be replaced with another string

```
>>> from re import sub
>>> text = '''
I love my iPad.
iPads are the best!
'''

>>> print(sub('iPad', 'Galaxy Tab', text))
```

```
I love my Galaxy Tab.
Galaxy Tabs are the best!
```

Backreferences

- Sometimes we need to relate one part of a regular expression to another
- For instance, how could we find which strings which appear twice in a hyphenated word like 'putt-putt' or 'tuk-tuk'?
 - Pattern `([a-z]+)-[a-z]+` is inadequate because it matches adjectives like 'well-fed'
- To solve this we need to refer back to a sub-pattern which has already been matched

tuk-tuk | 'tuktuk |

noun

(in Thailand) a three-wheeled motorized vehicle used as a taxi.

```
>>> text = '''  
I told my well-fed, overweight tuk-tuk  
driver to slow down!  
'''  
  
>>> findall('([a-z]+)-[a-z]+', text)  
['well', 'tuk']
```


Backreferences

- A *group* is marked in a regular expression by round brackets, `(...)`
- After a group within a pattern has been matched, it can be referred to later in the expression as `'\x'`, where `x` is a positive number
 - Regular expression groups are counted from one, not zero
- The hyphenated-word problem can therefore be solved with the following pattern:

`([a-z]+)-\1`

```
>>> text = '''  
I told my well-fed, overweight tuk-tuk  
driver to slow down!  
'''  
  
>>> findall(r'([a-z]+)-\1', text)  
['tuk']
```

Backreferencing in replacement text

- Combining substitution and backreferencing allows sophisticated forms of text manipulation

```
>>> text = '''
... We usually want to truncate monetary values
... like $45.6729 and $1.33333 to two decimal
... places while leaving other numbers like
... percentage 45.6792% alone.
... '''
>>> print(sub(' (\\$[0-9]+\\. [0-9]{2}) [0-9]*', r'\\1', text))

We usually want to truncate monetary values
like $45.67 and $1.33 to two decimal
places while leaving other numbers like
percentage 45.6792% alone.
```

Parsing: Beyond pattern matching

- *Parsing* is the process of analysing the syntactic structure of text
 - It assumes the text follows a particular *grammar*
- Like regular expressions, a ‘parser’ can be used to find particular textual elements
 - It also allows us to perform specific actions for each element found
- Python’s `HTMLParser` module provides the ability to parse HTML-based documents, and provides an alternative to using regular expressions
 - However, using this module requires some knowledge of ‘object-oriented’ programming
 - Regexs are sufficient for our purposes in this teaching unit

Before next week ...

- Complete this week's workshop exercises
- Work on Assessment Task 1B
- Practice with regular expressions as much as you can!