Searching & Matching using Regular Expressions
Regular Expressions (RE)

- Searching and extracting text is so common that Python has a very powerful library called *regular expressions* that handles many of these tasks quite elegantly.

- REs are a compact way to describe sophisticated patterns for matching and parsing strings of text.

- Entire books have been written on the topic of regular expressions. We will cover the basics here...

- For more detail on regular expressions, see:
  - [https://docs.python.org/2/library/re.html](https://docs.python.org/2/library/re.html)
Regular Expressions (RE)

A regular expression (regex) is a sequence of characters that define a search pattern, mainly for use in pattern matching with strings, or string matching, i.e. "find and replace"-like operations. The concept arose in the 1950s, when the American mathematician Stephen Kleene formalized the description of a regular language, and came into common use with the Unix text processing utilities `ed`, an editor, and `grep`, a filter.

In modern usage, regular expressions are often distinguished from the derived, but fundamentally distinct concept of `regex` pronunciation or `regexp`, which no longer describes a regular language. See below for details.

Regexp is so useful in computing that the various systems to specify regexes have evolved to provide both a basic and extended standard for the grammar and syntax:

- modern regex heavy: augment the standard. Regex processors are found in several search engines, search and replace dialogs of several word processors and text editors, and in the command lines of text processing utilities, such as `sed` and `AWK`.

Many programming languages provide regex capabilities, some built-in (for example `Perl`, `JavaScript`, `Ruby`, `AWK`, and `Tcl`) and others via a standard library (for example `.NET languages`, `Java`, `Python`, `POSIX C`, and `C++ since C++11`). Most other languages offer regexes via a library.
More on Regular Expressions

In computing, a regular expression, also referred to as “regex” or “regexp”, provides a concise and flexible means for matching strings of text, such as particular characters, words, or patterns of characters. A regular expression is written in a formal language that can be interpreted by a regular expression processor.

- Very powerful and quite cryptic – and compact
- Regular expressions are a language unto themselves
- A language of “marker characters” - programming with characters
- Finite Automate (FA) or Finite-State-Machines (FSM) are the ”machinery” behind REs
Regular Expressions

• How to “concisely” describe a string that denotes an integer value?
  – ‘0’, ‘245’, but not ‘abc’ or ‘0.1’
  – “one or more digit characters”…

• Using RE we could say that an integer is described as:

\[
\text{integer} = [0-9]^+
\]

• This is a ”simple” regular expression
• With its own “language”.
Regular Expressions for Integers

• Using RE we could say that an integer is described as:

integer = [0-9]+
Regular Expressions for Integers

- Using RE we could say that an integer is described as:

\[ \text{integer} = [0-9]^+ \]

One or more occurrences of one of these characters

The collection of characters ‘0’, ‘1’, … , ‘9’

NOTE: This is not the python regular expression syntax
Regular Expressions for Integers

• Using RE we could say that an integer is described as:

\[ \text{integer} = [0-9]+ \]

One or more occurrences of one of these characters

The collection of characters ‘0’, ‘1’, … , ‘9’

• So, this compact description, describes:

‘00001’, ‘12345’

• But not:

’FX01’, ‘1.2345’

**NOTE:** This is not the python regular expression syntax
Regular Expressions for Reals

• Using RE we could say that a real is described as:

\[ \text{real} = (\pm)?[0-9]+(\.)[0-9]+ \]

The ‘.’ character.
Either ‘+’ or ‘-’, if at all.

• So, this compact description describes:

‘-0.1’, ‘+1.2345’, ‘-12.345’

• But not:

’0.0.1’, ‘1.2AF4’

**NOTE:** This is not the python regular expression syntax
Python Regular Expressions Quick Guide

^  Matches at the beginning of a line
$  Matches at the end of the line
.  Matches any character (hence the need to use \. for the ‘.’ character
\s  Matches whitespace
\S  Matches any non-whitespace character
*  Repeats a character zero or more times
*? Repeats a character zero or more times (non-greedy)
+  Repeats a character one or more times
+? Repeats a character one or more times (non-greedy)
[aeiou] Matches a single character in the listed set
[^XYZ] Matches a single character not in the listed set
[a-z0-9] The set of characters can include a range
(  Indicates where string extraction is to start
)  Indicates where string extraction is to end
Regular Expressions Module

- Before you can use regular expressions in your program, you must import the library using "import re"

- You can use `re.search()` to see if a string matches a regular expression, similar to using the `find()` method for strings

- You can use `re.findall()` extract portions of a string that match your regular expression similar to a combination of `find()` and slicing: `var[5:10]`
Using `re.search()` like `find()`

```python
import re
hand = open('mbox-short.txt')
for line in hand:
    line = line.rstrip()
    if re.search('From:', line):
        print line
```
Using `re.search()` like `startswith()`

```python
hand = open('mbox-short.txt')
for line in hand:
    line = line.rstrip()
    if line.startswith('From:') >= 0:
        print line
```

```
import re

hand = open('mbox-short.txt')
for line in hand:
    line = line.rstrip()
    if re.search('^From:', line):
        print line
```

We fine-tune what is matched by adding special characters to the string
Wild-Card Characters (repetition)

- The dot character matches any character
- If you add the asterisk character, the character is “any number of times”

```
RE = ^X.*:  
```

- Matches ‘X’ at the start of the line only
- Any number of times
- Matches any character
- Ends with ‘?’

X–Sieve: CMU Sieve 2.3
X–DSPAM–Result: Innocent
X–DSPAM–Confidence: 0.8475
X–Content–Type–Message–Body: text/plain
Wild-Card Characters (repetition)

- The dot character matches any character
- If you add the asterisk character, the character is “any number of times”

\[ \text{RE} = ^\text{X}.*:\]

- Matches ‘X’ at the start of the line only
- Any number of times
- Matches any character
- Ends with ‘?’

X-Sieve: CMU Sieve 2.3
X-DSPAM-Result: Innocent
X-DSPAM-Confidence: 0.8475
X-Content-Type-Message-Body: text/plain
Fine-Tuning Your Match Example

• Depending on how “clean” your data is and the purpose of your application, you may want to narrow your match down a bit

  Matches ‘X’ at the start of the line only followed by the ‘-’ character

  One or more times

  RE = ^X-\S+:  

  Matches any non-whitespace character

  Ends with ‘?’

X-Sieve: CMU Sieve 2.3
X-DSPAM-Result: Innocent
X-Plane is behind schedule: two weeks one or more times
Matching and Extracting Data

- The `re.search()` returns a `True/False` depending on whether the string matches the regular expression.
- If we actually want the matching strings to be extracted, we use `re.findall()`.

```
>>> import re
>>> x = 'My 2 favorite numbers are 19 and 42'
>>> y = re.findall('[0-9]+', x)
>>> print(y)
['2', '19', '42']
```
Matching and Extracting Data

• When we use `re.findall()`, it returns a list of zero or more sub-strings that match the regular expression

```python
>>> import re
>>> x = 'My 2 favorite numbers are 19 and 42'
>>> y = re.findall('[0-9]+', x)
>>> print y
['2', '19', '42']
>>> y = re.findall('[AEIOU]+', x)
>>> print y
[]
```
What if a Pattern is a Prefix of Another?

- **String:** "I'm a lumberjack and I'm OK"

  \[\text{RE} = \'.*\'\]

- Sometimes we would like to match the longer string other times split it into two...

```python
>>> import re
>>> x = "I'm a lumberjack and I'm OK"
>>> y = re.findall('.*\'m',x)
>>> print y
>>> ['"I'm a lumberjack and I'm"]

>>> y = re.findall('.*?\'m',x)
>>> print y
>>> ['"I'm", " a lumberjack and I'm"]
```

Greedy

Non-Greedy
Understanding Greedy Matching

- The repetition characters (+ and *) “push” outwards in both directions to **greedily** match the longest string.

Matches ‘F’ at the start of the line only

```python
>>> import re
>>> x = 'From: Using the : character'
>>> y = re.findall('^F.+?:', x)
>>> print y
['From: Using the :']
```

- `RE = ^F.+?:`
  - Matches ‘F’ at the start of the line only
  - One or more times
  - Matches any non-whitespace character
  - Ends with ‘?’
Understanding Greedy Matching

• The repetition characters (+ and *) “push” outwards in both directions to **greedily** match the longest string

```python
>>> import re
>>> x = 'From: Using the : character'
>>> y = re.findall('^F.+?:', x)
>>> print y
['From: Using the :']
```

Matches ‘F’ at the start of the line only

- One or more times
- Matches any non-whitespace character
- Ends with ‘:’

Why didn’t it match: ['From:'] only?
Understanding Greedy Matching

• The qualifier ? In a RE means “match the first possible string”, i.e., non-greedily

```python
>>> import re
>>> x = 'From: Using the : character'
>>> y = re.findall('^F.+?:', x)
>>> print y
['From:']
```

RE = ^F.+?:

- Matches ‘F’ at the start of the line only
- One or more times but non-greedily
- Matches any non-whitespace character
- Ends with ‘?’
Understanding Greedy Matching

- The qualifier `?` in a RE means “match the first possible string”, i.e., non-*greedily*

```
>>> import re
>>> x = 'From: Using the : character'
>>> y = re.findall('^F.+:', x)
>>> print y
['From:']
```

- Matches ‘F’ at the start of the line only
- One or more times but non-greedily
- Matches any non-whitespace character
- Ends with ‘?’
- `RE = ^F.+?:`
Context in String Matching

- Often you can only match specific strings in the context of a string with other “elements”

- Example: matching sender e-mails in a text file

```python
>>> import re
>>> x = 'From stephen.marquard@uct.ac.za Sat Jan 5 09:14:16 2008'
>>> y = re.findall(r'\S+@\S+', x)
>>> print y
['stephen.marquard@uct.ac.za']
```

RE = \S+@\S+

At least one non-whitespace character
Context in String Matching

• This will match an e-mail anywhere in the text
  - Not only the sender’s e-mail…
  - Need to limit the context of matching
  - But capture only the e-mail as a string…

• Parenthesis will tell where to start and to end the actual matched string in the pattern

```python
>>> import re
>>> x = 'From stephen.marquard@uct.ac.za Sat Jan 5 09:14:16 2008'
>>> y = re.findall('^From: .*? (\S+@\S+)', x)
>>> print y
['stephen.marquard@uct.ac.za']
```

RE = `^From:.*?\(\S+@\S+)`

At least one non-whitespace character
RegEx vs. Slicing

• Extracting Hostname from an e-mail string

```python
>>> data = 'From stephen.marquard@uct.ac.za Sat Jan 5 09:14:16 2008'
>>> atpos = data.find('@')
>>> print atpos
21
>>> sppos = data.find(' ', atpos)
>>> print sppos
31
>>> host = data[atpos+1 : sppos]
>>> print host
uct.ac.za
```

Or

```python
>>> data = 'From stephen.marquard@uct.ac.za Sat Jan 5 09:14:16 2008'
>>> words = data.split()
>>> email = words[1]
>>> pieces = email.split('@')
>>> print pieces[1]
uct.ac.za
```
RegEx vs. Slicing

- The Regular Expression version

```python
>>> import re
>>> data = 'From stephen.marquard@uct.ac.za Sat Jan 5 09:14:16 2008'
>>> y = re.findall('@([^ ]*)',data)
>>> print y
['uct.ac.za']

RE=('@([^ ]*)')
RegEx vs. Slicing

• The Regular Expression version

```python
>>> import re
>>> data = 'From stephen.marquard@uct.ac.za Sat Jan 5 09:14:16 2008'
>>> y = re.findall('@([^ ]*)',data)
>>> print y
['uct.ac.za']
```

RE=`(@([^ ]*)'`

Look through the string until you find an @ sign
RegEx vs. Slicing

• The Regular Expression version

```python
>>> import re
>>> data = 'From stephen.marquard@uct.ac.za Sat Jan 5 09:14:16 2008'
>>> y = re.findall('@([^ ]*)', data)
>>> print y
['uct.ac.za']
```

But match only a non-blank character that follows and many of them

```
RE=( '@( [^ ]*)' )
```

Look through the string until you find an @ sign
RegEx vs. Slicing

• The Regular Expression version

```python
>>> import re
>>> data = 'From stephen.marquard@uct.ac.za Sat Jan 5 09:14:16 2008'
>>> y = re.findall('@([^ ]*)', data)
>>> print y
['uct.ac.za']
```

- But match only a non-blank character that follows
- and many of them

```python
RE=('@([^ ]*)',)
```

- Look through the string until you find an @ sign
- Finally, extract only these…
RegEx vs. Slicing

• How about the e-mails in the middle of the text
  - We want to skip those…
  - But still capture the ones that are sender e-mails…

```python
>>> import re
>>> data = 'From stephen.marquard@uct.ac.za Sat Jan 5 09:14:16 2008'
>>> y = re.findall('^From .*@([\^ ]*)', data)
>>> print y
['uct.ac.za']
```

RE=('^From .*@([\^ ]*)')

Match staring at the beginning of the line if you see ‘From’
RegEx vs. Slicing

- How about the e-mails in the middle of the text
  - We want to skip those…
  - But still capture the ones that are sender e-mails…

```python
>>> import re
>>> data = 'From stephen.marquard@uct.ac.za Sat Jan 5 09:14:16 2008'
>>> y = re.findall(r'^From .*@([^\ )]*)', data)
>>> print y
['uct.ac.za']
```

RE=${'^From .*@([^ ]*)'}

Skip all the characters before you detect a ‘@’
RegEx vs. Slicing

- How about the e-mails in the middle of the text
  - We want to skip those…
  - But still capture the ones that are sender e-mails…

```python
>>> import re
>>> data = 'From stephen.marquard@uct.ac.za Sat Jan 5 09:14:16 2008'
>>> y = re.findall('^From .*@([^ ]*)', data)
>>> print y
['uct.ac.za']

RE=('^From .*@([^ ]*)')
```

Start extracting
RegEx vs. Slicing

- How about the e-mails in the middle of the text
  - We want to skip those…
  - But still capture the ones that are sender e-mails…

```python
>>> import re
>>> data = 'From stephen.marquard@uct.ac.za Sat Jan 5 09:14:16 2008'
>>> y = re.findall(r'^From .*@([^ ]*)', data)
>>> print y
['uct.ac.za']
```

RE=('^From .*@([^ ]*)'  
Any number of non-blank character)
RegEx vs. Slicing

• How about the e-mails in the middle of the text
  - We want to skip those…
  - But still capture the ones that are sender e-mails…

```python
>>> import re
>>> data = 'From stephen.marquard@uct.ac.za Sat Jan 5 09:14:16 2008'
>>> y = re.findall('^From .*@([^ ]*)', data)
>>> print(y)
['uct.ac.za']
```

RE=('^From .*@([^ ]*)')

Until you see a blank character
A Special ‘Escape’ Character

• If you want a special regular expression character to just behave normally (most of the time) you prefix it with '\'

```python
>>> import re
>>> x = 'We just received $10.00 for cookies.'
>>> y = re.findall('\$[0-9.]+',x)
>>> print y
['$10.00']
```

RE=( '\$[0-9.]+' )

- A real dollar sign
- A digit or a period

At least one or more