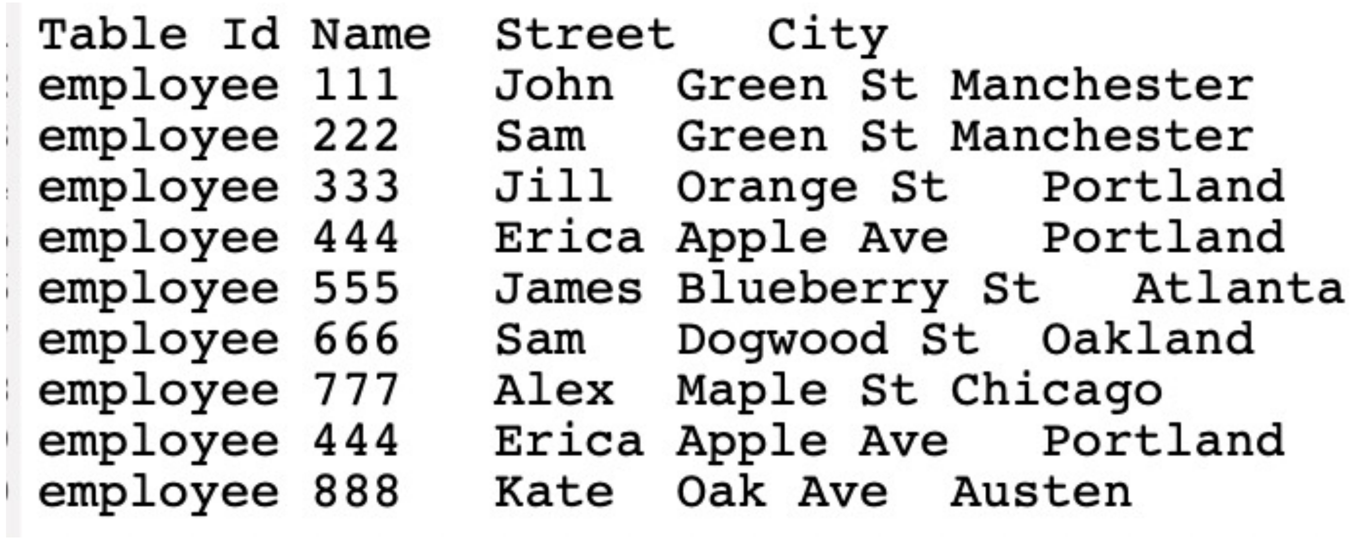
Regular Expressions

RegEx

Sometimes Data to be Imported Looks Like This

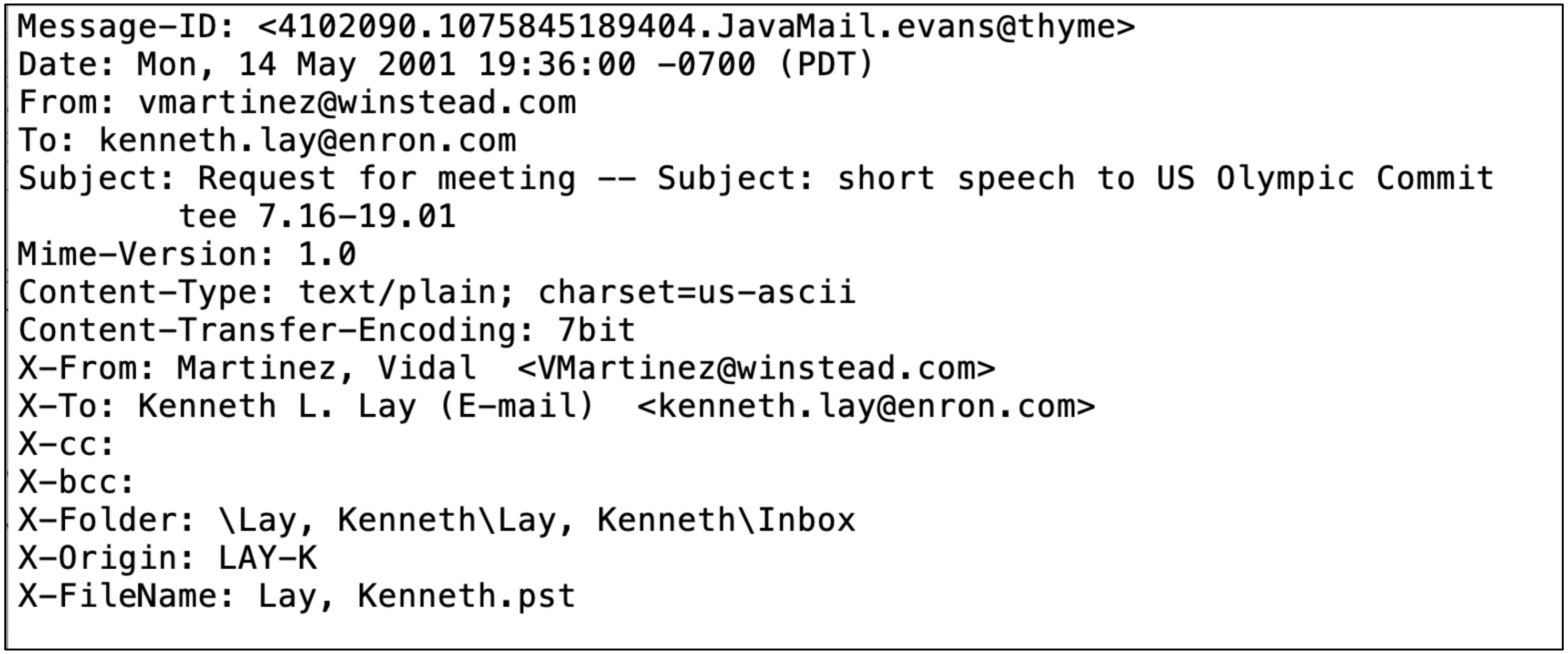


```
Table Id Name  Street   City
employee 111   John  Green St Manchester
employee 222   Sam   Green St Manchester
employee 333   Jill  Orange St   Portland
employee 444   Erica Apple Ave   Portland
employee 555   James Blueberry St   Atlanta
employee 666   Sam   Dogwood St  Oakland
employee 777   Alex  Maple St Chicago
employee 444   Erica Apple Ave   Portland
employee 888   Kate  Oak Ave  Austen
```

Slightly more complex data

```
>NM_001302688.2 Homo sapiens apolipoprotein E (APOE), transcript variant 1, mRNA
CTACTCAGCCCCAGCGGAGGTGAAGGACGTCCTTCCCCAGGAGCCGGTGAGAAGCGCAGTCGGGGGCACG
GGGATGAGCTCAGGGGCCTCTAGAAAGAGCTGGGACCCTGGGAACCCCTGGCCTCCAGACTGGCCAATCA
CAGGCAGGAAGATGAAGGTTCTGTGGGCTGCGTTGCTGGTCACATTCCTGGCAGGATGCCAGGCCAAGGT
GGAGCAAGCGGTGGAGACAGAGCCGGAGCCCGAGCTGCGCCAGCAGACCGAGTGGCAGAGCGGCCAGCGC
TGGGAACTGGCACTGGGTCGCTTTTGGGATTACCTGCGCTGGGTGCAGACACTGTCTGAGCAGGTGCAGG
AGGAGCTGCTCAGCTCCCAGGTCACCCAGGAACTGAGGGCGCTGATGGACGAGACCATGAAGGAGTTGAA
GGCCTACAAATCGGAACTGGAGGAACAACTGACCCCGGTGGCGGAGGAGACGCGGGCACGGCTGTCCAAG
GAGCTGCAGGCGGCGCAGGCCCGGCTGGGCGCGGACATGGAGGACGTGTGCGGCCGCCTGGTGCAGTACC
GCGGCGAGGTGCAGGCCATGCTCGGCCAGAGCACCGAGGAGCTGCGGGTGCGCCTCGCCTCCCACCTGCG
CAAGCTGCGTAAGCGGCTCCTCCGCGATGCCGATGACCTGCAGAAGCGCCTGGCAGTGTACCAGGCCGGG
GCCCGCGAGGGCGCCGAGCGCGGCCTCAGCGCCATCCGCGAGCGCCTGGGGCCCCTGGTGGAACAGGGCC
GCGTGCGGGCCGCCACTGTGGGCTCCCTGGCCGGCCAGCCGCTACAGGAGCGGGCCCAGGCCTGGGGCGA
GCGGCTGCGCGCGCGGATGGAGGAGATGGGCAGCCGGACCCGCGACCGCCTGGACGAGGTGAAGGAGCAG
GTGGCGGAGGTGCGCGCCAAGCTGGAGGAGCAGGCCCAGCAGATACGCCTGCAGGCCGAGGCCTTCCAGG
CCCGCCTCAAGAGCTGGTTCGAGCCCCTGGTGGAAGACATGCAGCGCCAGTGGGCCGGGCTGGTGGAGAA
GGTGCAGGCTGCCGTGGGCACCAGCGCCGCCCCTGTGCCCAGCGACAATCACTGAACGCCGAAGCCTGCA
GCCATGCGACCCCACGCCACCCCGTGCCTCCTGCCTCCGCGCAGCCTGCAGCGGGAGACCCTGTCCCCGC
CCCAGCCGTCCTCCTGGGGTGGACCCTAGTTTAATAAAGATTCACCAAGTTTCACGCA
```

Most input files are not nice and tidy: they’re messy



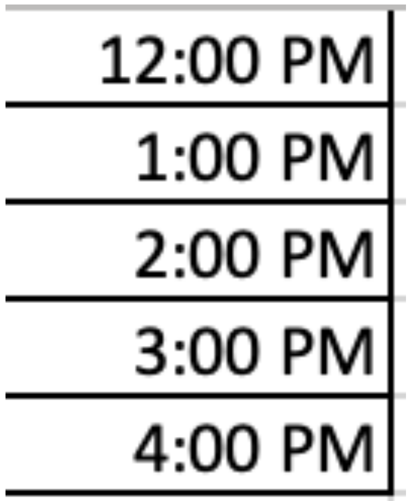
```
Message-ID: <4102090.1075845189404.JavaMail.evans@thyme>
Date: Mon, 14 May 2001 19:36:00 -0700 (PDT)
From: vmartinez@winstead.com
To: kenneth.lay@enron.com
Subject: Request for meeting -- Subject: short speech to US Olympic Commit
        tee 7.16-19.01
Mime-Version: 1.0
Content-Type: text/plain; charset=us-ascii
Content-Transfer-Encoding: 7bit
X-From: Martinez, Vidal  <VMartinez@winstead.com>
X-To: Kenneth L. Lay (E-mail)  <kenneth.lay@enron.com>
X-cc:
X-bcc:
X-Folder: \Lay, Kenneth\Lay, Kenneth\Inbox
X-Origin: LAY-K
X-FileName: Lay, Kenneth.pst
```

Complex Fields

Sometimes we find that multiple bits of data are combined into a single field

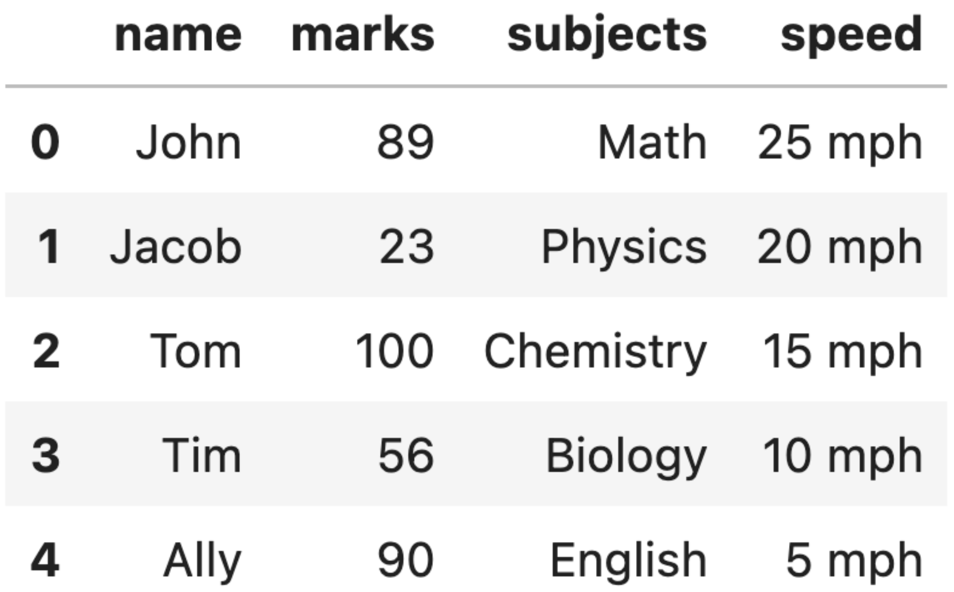
```
HUMAN|HGNC=4242|UniProtKB=O94808
```

Data may have extraneous characters



12:00 PM
1:00 PM
2:00 PM
3:00 PM
4:00 PM

PM



	name	marks	subjects	speed
0	John	89	Math	25 mph
1	Jacob	23	Physics	20 mph
2	Tom	100	Chemistry	15 mph
3	Tim	56	Biology	10 mph
4	Ally	90	English	5 mph

mph

The problem continued….

- Or we only want to load some of the information, and not the rest
  - Example, a field contains

        Student Id: 12345

    And we only want the number

Need a way to parse out the information we want

Tests for regular string-matching in python

- Can directly compare two strings

```
if (“Jon” == “Jon”)
        This is true since they are exactly the same

if (“Jon” == “Here is a string with the name Jon in it”)
        False, since they’re not exactly the same
```

- Can test to see if the substring exists

```
If (“Jon” in “Here is a string with the name Jon in it”)

True since Jon is a substring
```

Tests for regular string-matching in python

- .index = returns the position of a substring in a string

  S = ”Here is a string with the name Jon in it”

  S.index(“Jon”)

  31   = the position of the substring in S

How does regular string-matching work?

Linear search looking for an exact match

Jon
Here is a string with the name Jon in it.

Ambiguity

- What if we want to look for either Jon or Jan?
- Can either explicitly code for both or include ambiguity into the search.
- Do this with regular expressions: [ao] = match either an “a” or an “o”

```
J[ao]n
Here is a string with the name Jon in it.
```

```
J[ao]n
Here is a string with the name Jan in it.
```

Ambiguity with [ ]

- Any letters, numbers or symbols inside of the [ ] will be included in the search.
- Only matches 1 of them at a time (we'll fix this later)
- Can use ranges

  [A-Z] = all upper case letters

  [a-z] = all lower case letters

  [0-9] = all digits

  [A-Za-z0-9] = all letters and digits

More Ambiguity

- What if we want to look for either Jon or Jan or any 3-lettered word starting with an uppercase J and ending with an n?
- \w matches any word character (letter) A-Za-z or the digits 0-9 or an underscore “_”

```
J\wn
Here is a string with the name Jon in it.

J\wn
Here is a string with the name Jan in it.
```

Regular Expressions (RegEx or regex)

- Regular expressions give us a way to match **patterns** of text and to retrieve the text.

- Can allow for ambiguity
- Can allow for repetition
- Can be greedy when matching (the default), or not

How to Create a Pattern – A Starting Point

- Look for characters that the strings to matched have in common.
  - Are they in the same place in the strings? Use these in the pattern
  - Is the space between them variable?
    - May have to use multiplicity with the ambiguity codes

- Look at what's different
  - Are the characters restricted to a subset of characters?
    - [ ] is a good way to go
  - Can any character appear?
    - Something like \w or \S (or others) could be a good way to go
  - How long are the stretches of difference?
    - Might have to use multiplicity (e.g. + or *)

The most used python command

```
import re
```

- `re.findall(pattern, string)`
  - finds all matches of the pattern in the string and returns a list of all of the matches (even if there's just one)

- re also has a split function
  - similar to the regular .split, but allows the use of patterns

- There is a search function
  - returns a "match object" that then has to be processed.

findall

```
import re

s = “Here is a string with the name Jon in it.”

m = re.findall(“Jon”, s)

[‘Jon’]
```

findall returns a list of **ALL** of the matches to the pattern

```
s = “Here is a string with the name Jon and Jonathon in it”

m = re.findall(“Jon”, s)

[‘Jon’, ’Jon’]
```

findall

```
import re

s = “Here is a string with the name Jon in it.”

m = re.findall(“J[oa]n”, s)

[‘Jon’]

s = “Here is a string with the name Jan in it.”

m = re.findall(“J[oa]n”, s)

[‘Jan’]
```

findall

```
import re

s = “Here is a string with the name Jon in it.”

m = re.findall(“J\wn”, s)

[‘Jon’]

s = “Here is a string with the name Jan and Jon in it.”

m = re.findall(“J\wn”, s)

[‘Jan’, ‘Jon’]
```

findall

```
import re

s = “Here is a string with the name Jon in it.”

m = re.findall(“J.n”, s)
```

. matches ANY character

```
[‘Jon’]

s = “Here is a string with the name Jon in it.”

m = re.findall(“J\Sn”, s)
```

\S matches ANY non-white-space character

```
[‘Jon’]
```

But what about…

```
import re

s = “Here is a string with the name Jooon in it.”

m = re.findall(“J\wn”, s)

[]        empty list – no matches
```

Multiplicity

+ = matches 1 or more

    \w+ matches 1 or more word characters

* = matches 0 or more

    \w* matches 0 or more word characters

{n} matches exactly n

    \w{3} matches exactly 3 word characters

{n, m} matches at least n and at most m

    \w{3,5} matches 3, 4, or 5 word characters

Can be used with any regex ambiguity codes including [ ]

But what about…

```
import re

s = “Here is a string with the name Jooon in it.”

m = re.findall(“J\w+n”, s)

[’Jooon’]

s = “Here is a string with the name Jooon and Joan in it.”

m = re.findall(“J\w+n”, s)

[’Jooon’, ‘Joan’]
```

Could also do…

```
import re

s = “Here is a string with the name Jooon in it.”

m = re.findall(“J\w*n”, s)

[’Jooon’]

s = “Here is a string with the name Jooon and Joan in it.”

m = re.findall(“J\S+n”, s)

[’Jooon’, ‘Joan’]
```

Could also do…

```
import re

s = “Here is a string with the name Jooon in it.”

m = re.findall(“J[oa]+n”, s)

[’Jooon’]

s = “Here is a string with the name Jooon and Joan in it.”

m = re.findall(“J[oa]*n”, s)

[’Jooon’, ‘Joan’]
```

Can Mix and Match

```
import re

s = “Here is a string with the name Jooon in it.”

m = re.findall(“J[oa]+\w+n”, s)

[’Jooon’]

s = “Here is a string with the name Jooon and Joan in it.”

m = re.findall(“J[oa]\w+n”, s)

[’Jooon’, ‘Joan’]
```

Ambiguity Codes

- a, X, 9, <
  - ordinary characters just match themselves exactly.
  - The meta-characters which do not match themselves because they have special meanings are:

    . ^ $ * + ? { [ ] \ | ( )

    (details later)

- \w
  - lowercase w
  - matches a "word" character: a letter or digit or under score [a-zA-Z0-9_].
  - Note that although "word" is the mnemonic for this, it only matches a single word char, not a whole word. \W (upper case W) matches any non-word character.

Ambiguity Codes

- \d
  - decimal digit [0-9] (some older regex utilities do not support \d, but they all support \w and \s)
- \s
  - lowercase s matches a single whitespace character -- space, newline, return, tab, form [\n\r\t\f].
- \S
  - (upper case S) matches any non-whitespace character.
- \t, \n, \r
  - tab, newline, return

Ambiguity Codes

- . (a period)
  - matches any single character except newline '\n’
- \b
  - boundary between word and non-word

Ambiguity Codes

- ^
  - Match the pattern starting at the start of the string
    - Can’t have anything before it

```
import re

s = “Jon is in a string”

m = re.findall(“^J\wn”, s)

[Jon]
```

Ambiguity Codes

- $
  - Match the pattern ending at the end of the string
    - Can’t have anything before it

```
import re

s = “Here is a string with Jon”

m = re.findall(“J\wn$”, s)

[Jon]
```

Ambiguity Codes

- \
  - ”Escape” character
  - Inhibit the "specialness" of a character.
  - For example, use \. to match a period or \\ to match a slash. If you are unsure if a character has special meaning, such as '@', you can try putting a slash in front of it, \@. If its not a valid escape sequence, like \c, your python program will halt with an error.
  - Put \ before any of these . ^ $ * + ? { [ ] \ | ( ) if you want to match to them

```
s = “Here is a string with {Jon}”

m = re.findall(“\{J\wn\}”, s)

[{Jon}]
```

Examples

```
import re

s = “Jan met Jon”

m = re.findall(“^J\w+n”, s)          Matches at the start of s

[‘Jan’]

m = re.findall(“J\w+n$”, s)          Matches at the end of s

[‘Jon’]
```

Greediness

The pattern matching is greedy, it tries to match as much as possible of the string

```
s = “Jan met Jon”

m = re.findall(“J.+n”, s)
```

. (period) matches ANY character, including spaces

```
[‘Jan met Jon’]
```

Greediness

? = Makes the match ungreedy

s = “Jan met Jon”

m = re.findall(“J.+?n”, s)

[‘Jan’, ‘Jon’]

Capturing Parts of a Pattern

- Up to now, we've been retrieving the entire part of the string that matches the pattern
- We can match on a pattern, and capture only a part (or parts) of it
  - Handy if a pattern bigger than the substring of interest is needed to better identify the substring we want

```
Hello, my name is Professor Izmirli
Hello, my name is Professor Parker
Hello, my name is Professor Peitzsch
Hello, my name is Professor Douglas
```

Capturing Parts of a Pattern

Put parentheses around portion(s) of the pattern that you want captured

```
Hello, my name is Professor Izmirli
Hello, my name is Professor Parker
Hello, my name is Professor Peitzsch
Hello, my name is Professor Douglas
```

```
for line in name_list_above:
        m = re.findall(“Professor (\w+)”, line)
        m


[‘Izmirli’]
[‘Parker’]
[‘Peitzsch’]
[‘Douglas’]
```

Retrieving Parts of a Pattern

Can have multiple components captured

Hello, my name is Professor Izmirli and my office number is 1
Hello, my name is Professor Parker and my office number is 2
Hello, my name is Professor Peitzsch and my office number is
Hello, my name is Professor Douglas and my office number is 4

```
for line in name_list_above:
        m = re.findall(“Professor (\w+) .* number is (\d+)”,
line)
        m
```

```
[‘Izmirli’, ‘1’]
[‘Parker’, 2]
[ ]
[‘Douglas’, 4]
```

Notice there’s no match since the office number is missing

Summary

- There is a LOT more to regular expressions

- 99.9% of what you’ll need is in this slide deck

End of Regular Expressions