SI 601

Text Encodings and Finding Patterns in Text with Regular Expressions

Chris Teplovs (cteplovs@umich.edu)

Lead Developer, Digital Innovation Greenhouse

Adjunct Lecturer, School of Information

Some material courtesy of: Kevyn Collins-Thompson, Yuhang Wang, Qiaozhu Mai, Charles Severance, Patrick Dudas

Announcements

- MOOC beta testing opportunity (see Canvas)
- MIDAS Annual Symposium
  - "Big Data: Advancing Science, Changing the World"

Let’s Talk About Assignments

How many hours you spent working on assignments

8

5 4 10

6 1 3

20 4

Higher or Lower?

Extension: Just this time

- I am willing to extend the deadline for Lab and Homework #1 to Monday, 1:00pm
- If you have already submitted your assignment, you will receive a 5% point bonus, up to the maximum of the assignment
  - For example, on the Lab, which is worth 20 points, if you submitted by the deadline and you would have received 19 points, you will get 20. If you would have gotten 20, you will still get 20
  - For example, on the Homework, which is worth 100 points, if you submitted by the deadline and you would have received 85/100 you would get 90/100
- This is mostly to accommodate late enrollment in a way that does not untowardly penalize those who completed the assignment on time
- No further extensions will be granted: you may use your own “free” late days if you want
- This is only for Lab #1 and Homework #1

Lab & Homework Grading

- SungJin Nam will grade Lab and Homework Assignments
- Grades will be posted in about a week
- Grade questions: e-mail SungJin (sjnam@umich.edu) cc: cteplovs@umich.edu

Individual project: 1-page proposal

- Initial one-page proposal due Friday, September 23rd, 1:00 pm
- 20% of project grade (Overall, project is worth 30% of grade)
- One page
- I will provide feedback by the following class, and may meet in person.
- Proposal Guidelines (60 points):
    1. (10 points) Summarize and motivate your proposed project.
    2. (20 points) Choose and describe two different datasets.
    3. (20 points) Describe how you might manipulate and join the two datasets.
    4. (10 points) Describe one interesting visualization that you might perform with the joined data.
- You can propose an alternative project structure with prior approval

Piazza

- Ask questions, get help!

SI 601 Data Manipulation: Class Schedule

(Some details may change)

Date	Topic	Assignments Due (before start of class)
Sep 9	Course introduction Basics of Programming with Python	Install software as described in welcome email
Sep 16	**Text Processing and Pattern Extraction with Regular Expressions**	**Homework 1, Lab 1**
Sep 23	Fetching and Parsing Web content: HTML, JSON, XML	Homework 2, Lab 2 1-page Project Proposal Due
Sep 30	Fetching data from Large Online Services Querying data in a SQL Database	Homework 3, Lab 3
Oct 7	Large-scale data manipulation with MapReduce and Hadoop	Homework 4, Lab 4
Oct 14	Advanced topics: learning analytics, synthetic data	Homework 5, Lab 5
Oct 21	Course Review, Final project presentations	Project report due

Previous class: Python basics

- Strings, numbers
- Lists, sets, and dictionaries
- Sorting
- Basic control flow
- File read/write

<u>This class</u>:

Building on this, to call the python **re** regular expression module to find and extract text patterns

Text is a fundamental data type
Text encodes language and other critical information types

- Text (unstructured)
- Structured text
- Structured data (+ text)
- Graphics: images
- Video, multimedia

Understanding <u>text encoding and pattern matching</u> is critical to data manipulation.

- Log file analysis: extract IP addresses, user IDs, etc.
- Parsing and extracting text fields for import to a database.
- Cleaning or normalizing text for further processing.

Today's Class Roadmap

1. Text encodings and Unicode
2. Regex: Wildcards and other basics
3. Regex: Sets, ranges and alternatives
4. Regex: Advanced operations

Lab 2: Regular expressions

So you're handed a huge text file… how should you interpret its contents?

Increasing levels of structure

- Storage medium (physical bits: disk, memory)
- Character encoding: 8 bits → 1 letter*
- Character meaning: letters → words

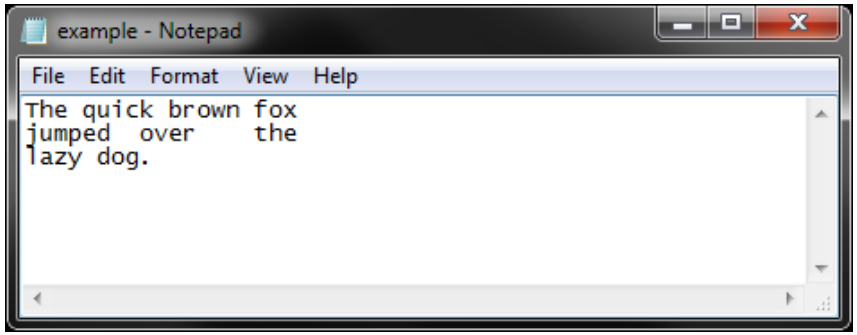
Today's lecture

- Delimited stream: lines, documents
- Structured content: tags, fields

Next lecture

- Files:
  - Individual
  - Archive (.zip, .gz, ….)
- Directories, collections

How is plain text stored in a file?



The text is encoded as a stream of numbers. Each number represents a letter, symbol, or special character like tab or space.

Byte 00	84	104	101	32	113	117	105	99	107	32	98	114	111	119	110	32
	T	**h**	**e**		**q**	**u**	**i**	**c**	**k**		**b**	**r**	**o**	**w**	**n**	
Byte 16	102	111	120	10	106	117	109	112	101	100	9	111	118	101	114	9
	f	**o**	**x**	**\n**	**j**	**u**	**m**	**p**	**e**	**d**	**\t**	**o**	**v**	**e**	**r**	**\t**
Byte 32	116	104	101	10	108	97	122	121	32	100	111	103	46	10		
	t	**h**	**e**	**\n**	**l**	**a**	**z**	**y**		**d**	**o**	**g**	**.**	**\n**		

<u>Delimiter:</u> a sequence of one or more characters used to specify the boundary between separate, independent regions in plain text or other data streams

Special whitespace *delimiters*:

`\t`	Tab = 9
`\n`	End-of-line = 10
`' '`	Space = 32

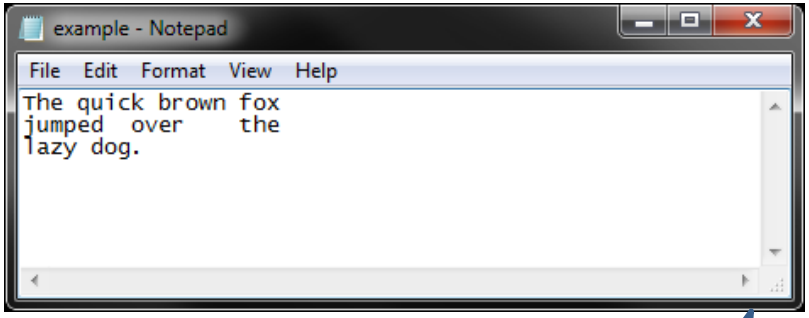
Who decided this?
Enter the ASCII encoding from 1963

Dec	Hx	Oct	Char	Dec	Hx	Oct	Html	Chr	Dec	Hx	Oct	Html	Chr	Dec	Hx	Oct	Html	Chr
0	0	000	NUL (null)	32	20	040		Space	64	40	100	@	@	96	60	140	`	`
1	1	001	SOH (start of heading)	33	21	041	!	!	65	41	101	A	A	97	61	141	a	a
2	2	002	STX (start of text)	34	22	042	"	"	66	42	102	B	B	98	62	142	b	b
3	3	003	ETX (end of text)	35	23	043	#	#	67	43	103	C	C	99	63	143	c	c
4	4	004	EOT (end of transmission)	36	24	044	$	$	68	44	104	D	D	100	64	144	d	d
5	5	005	ENQ (enquiry)	37	25	045	%	%	69	45	105	E	E	101	65	145	e	e
6	6	006	ACK (acknowledge)	38	26	046	&	&	70	46	106	F	F	102	66	146	f	f
7	7	007	BEL (bell)	39	27	047	'	'	71	47	107	G	G	103	67	147	g	g
8	8	010	BS (backspace)	40	28	050	(	(	72	48	110	H	H	104	68	150	h	h
9	9	011	TAB (horizontal tab)	41	29	051	)	)	73	49	111	I	I	105	69	151	i	i
10	A	012	LF (NL line feed, new line)	42	2A	052	*	*	74	4A	112	J	J	106	6A	152	j	j
11	B	013	VT (vertical tab)	43	2B	053	+	+	75	4B	113	K	K	107	6B	153	k	k
12	C	014	FF (NP form feed, new page)	44	2C	054	,	,	76	4C	114	L	L	108	6C	154	l	l
13	D	015	CR (carriage return)	45	2D	055	-	-	77	4D	115	M	M	109	6D	155	m	m
14	E	016	SO (shift out)	46	2E	056	.	.	78	4E	116	N	N	110	6E	156	n	n
15	F	017	SI (shift in)	47	2F	057	/	/	79	4F	117	O	O	111	6F	157	o	o
16	10	020	DLE (data link escape)	48	30	060	0	0	80	50	120	P	P	112	70	160	p	p
17	11	021	DC1 (device control 1)	49	31	061	1	1	81	51	121	Q	Q	113	71	161	q	q
18	12	022	DC2 (device control 2)	50	32	062	2	2	82	52	122	R	R	114	72	162	r	r
19	13	023	DC3 (device control 3)	51	33	063	3	3	83	53	123	S	S	115	73	163	s	s
20	14	024	DC4 (device control 4)	52	34	064	4	4	84	54	124	T	T	116	74	164	t	t
21	15	025	NAK (negative acknowledge)	53	35	065	5	5	85	55	125	U	U	117	75	165	u	u
22	16	026	SYN (synchronous idle)	54	36	066	6	6	86	56	126	V	V	118	76	166	v	v
23	17	027	ETB (end of trans. block)	55	37	067	7	7	87	57	127	W	W	119	77	167	w	w
24	18	030	CAN (cancel)	56	38	070	8	8	88	58	130	X	X	120	78	170	x	x

0000000	84	104	101	32	113	117	105	99	107	32	98	114	111	119	110	32
	T	**h**	**e**		**q**	**u**	**i**	**c**	**k**		**b**	**r**	**o**	**w**	**n**	
0000016	102	111	120	10	106	117	109	112	101	100	9	111	118	101	114	9
	f	**o**	**x**	**\n**	**j**	**u**	**m**	**p**	**e**	**d**	**\t**	**o**	**v**	**e**	**r**	**\t**
0000032	116	104	101	10	108	97	122	121	32	100	111	103	46	10		
	t	**h**	**e**	**\n**	**l**	**a**	**z**	**y**		**d**	**o**	**g**	**.**	**\n**		

* ASCII = *American* Standard Code for Information Interchange

Be aware: A text file created on one platform (e.g. Windows) may use slightly different convention than another platform (e.g. Unix)



File saved with Windows/MS-DOS convention:

```
0000000   84  104  101   32  113  117  105   99  107   32   99  114  111  119  110   32
           T    h    e         q    u    i    c    k         b    r    o    w    n
0000016  102  111  120   13   10  112  117  109  112  101  116   99  118  101  114
           f    o    x   \r   \n    j    u    m    p    e    t    c    v    e    r
0000032    9  116  104  101   13   10   97  116  116  101  114   11  103   46   13
          \t    t    h    e   \r   \n    a    t    t    e    r   \v    g    .   \r
0000048   10
          \n
```

File saved with Unix convention:

```
0000000   84  104  101   32  113  105  105   99  107   32   99  111  119  110   32
           T    h    e         q    u    i    c    k         b    o    w    n
0000016  102  111  120   10  106  117  109  112  101  116   99  111  101  114    9
           f    o    x    \n    j    u    m    p    e    t    c    o    e    r   \t
0000032  116  104  101   10  108   97  122  121   32  100  111  103   46   10
           t    h    e    \n    l    a    z    y         d    o    g    .   \n
```

input_file =
open('inputfile.txt', 'rU')

A character set specifies how numbers should be interpreted as character symbols

- ASCII
  - 7 bits per character (0-127) = 1 byte
  - Since 1960, from telegraph codes
- ISO-Latin-1 (ISO-8859-1)
  - “Code page”
  - 8-bit (0-255) = 1 byte
  - Superset of ASCII
  - Basis for original Web standard for HTTP and HTML
  - High-bit characters add e.g. accented characters for most ‘Western’ languages
  - Microsoft Windows ANSI similar variant

Char	Code	Name	Description
à	224	agrave	a grave
á	225	aacute	a acute
â	226	acirc	a circumflex
ã	227	atilde	a tilde
ä	228	auml	a umlaut
å	229	aring	a ring
æ	230	aelig	ae ligature
ç	231	ccedil	c cedilla
è	232	egrave	e grave
é	233	eacute	e acute
ê	234	ecirc	e circumflex
ë	235	euml	e umlaut
ì	236	igrave	i grave
í	237	iacute	i acute
î	238	icirc	i circumflex
ï	239	iuml	i umlaut

Char	Code	Name	Description
ð	240	eth	eth
ñ	241	ntilde	n tilde
ò	242	ograve	o grave
ó	243	oacute	o acute
ô	244	ocirc	o circumflex
õ	245	otilde	o tilde
ö	246	ouml	o umlaut
÷	247	divide	division sign
ø	248	oslash	o slash
ù	249	ugrave	u grave
ú	250	uacute	u acute
û	251	ucirc	u circumflex
ü	252	uuml	u umlaut
ý	253	yacute	y acute
þ	254	thorn	thorn
ÿ	255	yuml	y umlaut

The last 32 characters of the ISO-Latin-1 encoding

There is no such thing as plain text.*

- We live in a multilingual world
- It doesn't make sense to have a string without knowing what encoding it uses.
- To deal with textual data, you first have to know how to decode the text!
- Every working programmer must know the basics of character sets, encodings.
- Enter… Unicode.

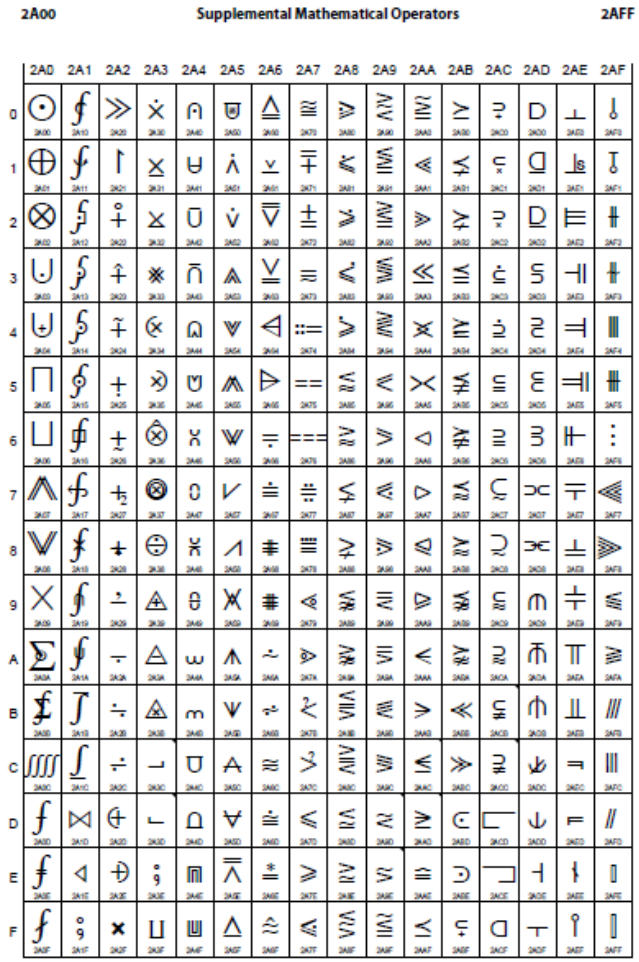
*Source: http://www.joelonsoftware.com/articles/Unicode.html

Unicode

- Encoding standard that covers more than 110,000 characters and more than 100 human languages.
  - Contains ISO-Latin-1: first 256 characters the same
  - First draft standard 1991: owned by Unicode Consortium
  - Beginning to replace ASCII and ISO character sets
- In theory, 0x0 to 0x10FFFF (1,114,112 characters)
  - Almost every language and past/present symbol you can think of
    - Coming soon: Star Trek languages, e.g. Klingon: private use area
- Implemented in most modern operating systems programming languages, and software
  - Including XML, Java, .NET framework, etc.

Source: http://www.joelonsoftware.com/articles/Unicode.html

In case you needed convincing that Unicode is serious about being a universal encoding…



[Source: http://www.unicode.org/charts/PDF/U1B00.pdf http://www.unicode.org/charts/PDF/U2A00.pdf]

Unicode support in a correctly-implemented Web browser

Azerbaijan (Latin script)	Heydar Aliyev (president)	Azərbaycan	Heydər Əliyev
Azerbaijan (Cyrillic script)	Heydar Aliyev (president)	Азәрбајҹан	Һејдәр Әлијев
Belgium (Flemish)	Rene Magritte (painter)	België	René Magritte
Belgium (French)	Rene Magritte (painter)	Belgique	René Magritte
Belgium (German)	Rene Magritte (painter)	Belgien	René Magritte
Bengal	Sukumar Ray	বাংলা	সুকুমার রায়
Bhutan	Gonpo Dorji (film actor)	འབྲུག་ཡུལ།	མགོན་པོ་རྡོ་རྗེ།
Cambodia (Khmer)	Venerable PreahBuddhaghosachar Chuon Nath	ខ្មែរ	ព្រះពុទ្ធឃោសាចារ្យ ជួន ណាត
Canada	Celine Dion (singer)	Canada	Céline Dion
Canada - Nunavut (Inuktitut language)	Susan Aglukark (singer)	ᓄᓇᕗᒻᒥᐅᑦ	ᓱᓴᓐ ᐊᒡᓗᒃᑲᖅ
Southeast USA (Cherokee Nation)	Sequoyah (invented syllabary)	ᏣᎳᎩ (Tsalagi)	ᏍᏆᏂᏙᏰᏍᏗ
People's Rep. of China	ZHANG Ziyi (actress)	中国	章子怡
People's Rep. of China	WONG Faye (singer)	中国	王菲
Czechia (Czech Republic)	Antonin Dvorak (composer)	Česko (Česká republika)	Antonín Dvořák
Denmark	Soren Hauch-Fausboll	Danmark	Søren Hauch-Fausbøll
Denmark	Soren Kierkegaard (theologian 1813-1855)	Danmark	Søren Kierkegård
Egypt (Masr)	Abdel Halim Hafez (singer)	مصر	عبدالحليم حافظ
Egypt (Masr)	Om Kolthoum (singer)	مصر	أم كلثوم
Eritrea	Berhane Zeray	ኤርትራ	ብርሃነ ዘርኣይ
Ethiopia	Haile Gebreselassie (Fastest man)	ኢትዮጵያ	ኃይሌ ገብረሥላሴ

Unicode characters: 'code points'

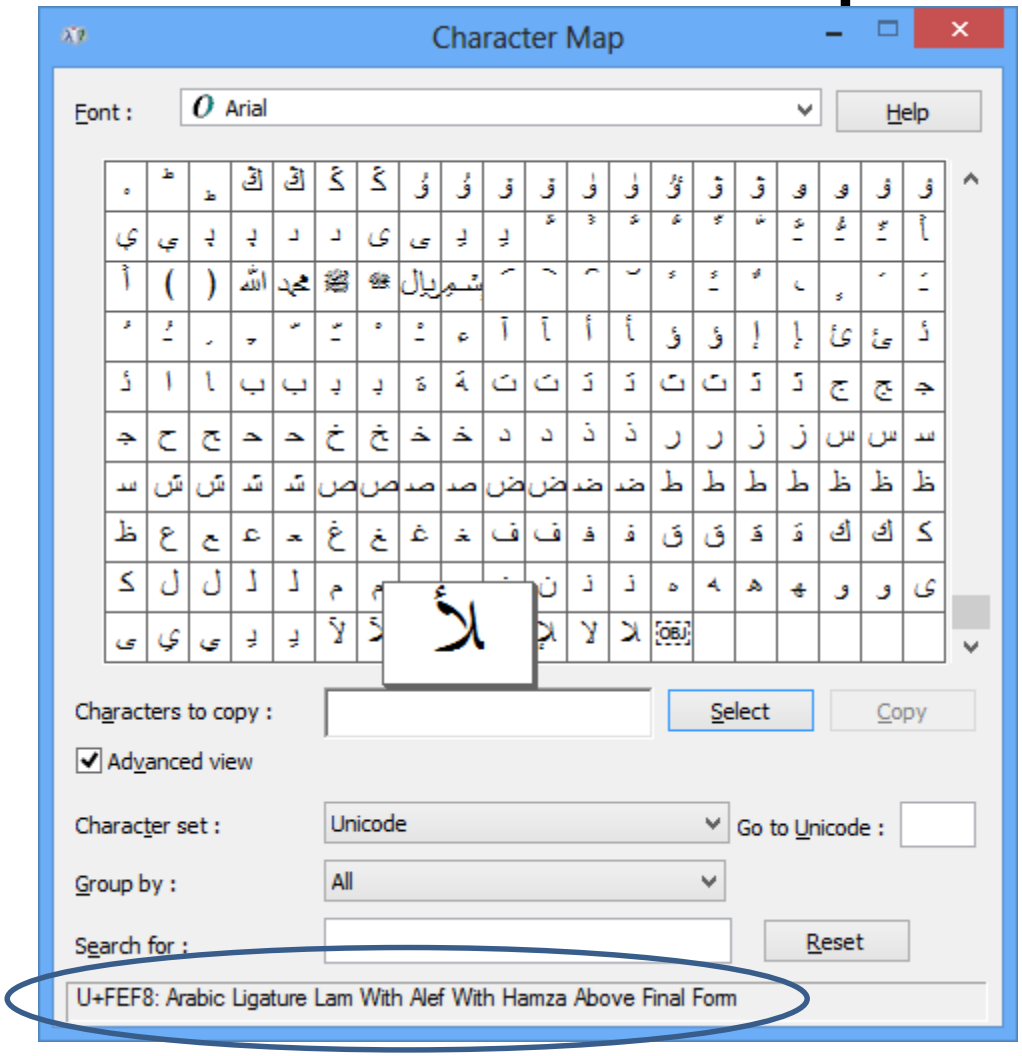
- Every letter in every alphabet is assigned a magic number by the Unicode consortium, like this: **U+0639**. This magic number is called a *code point*.
- The U+ means "Unicode" and the numbers are hexadecimal (base 16)
- They're all listed on the Unicode web site. (charmap utility in Windows)

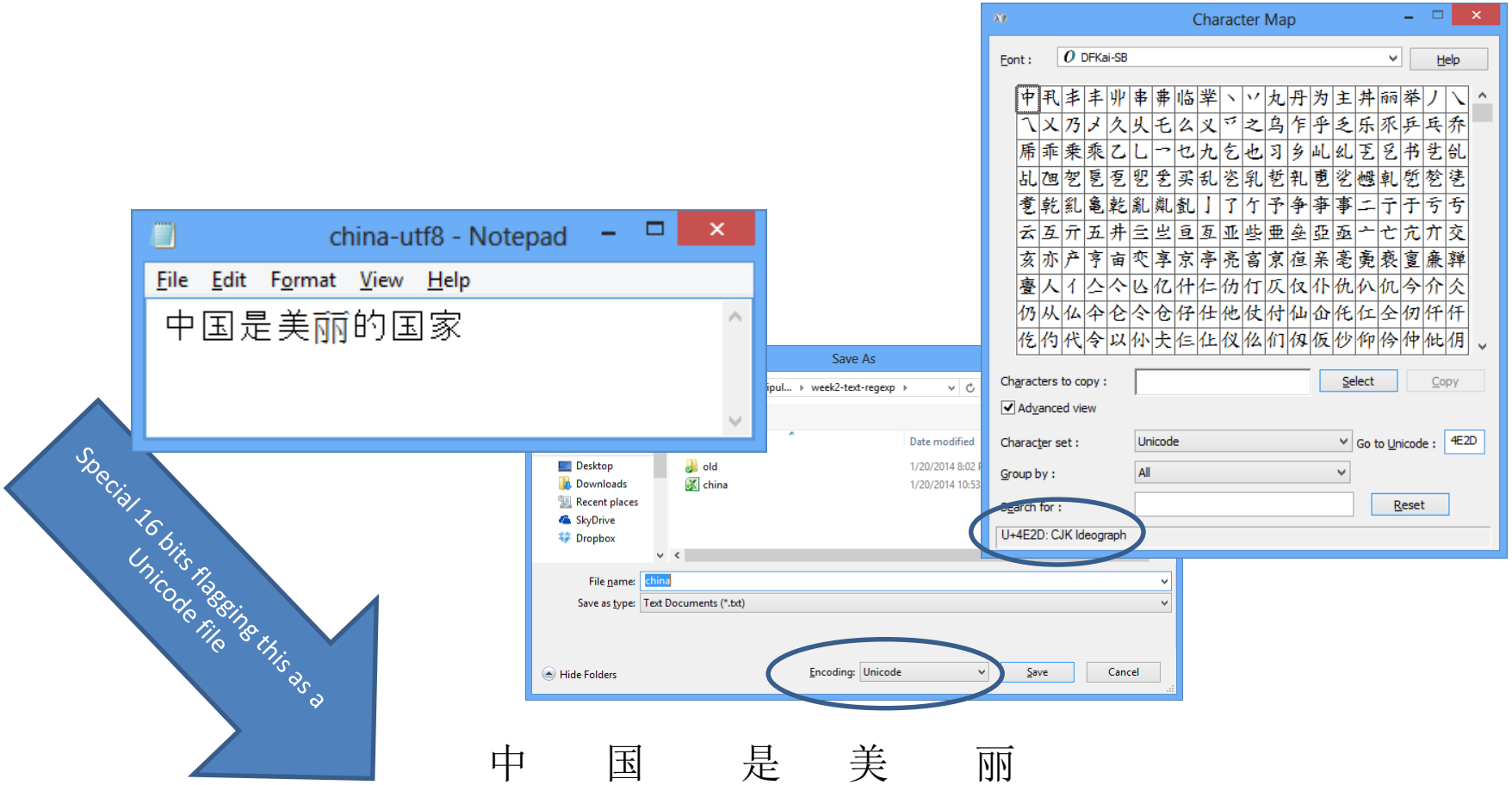
HELLO
U+0048 U+0065 U+006C U+006C U+006F

- Unicode can be stored in a file or string in many different ways, depending on efficiency considerations.
- UTF-8 Encoding:
  - In UTF-8, every code point from 0-127 is stored *in a single byte*.
  - Only code points 128 and above are stored using 2, 3, in fact, up to 6 bytes.
- Neat side effect: English text looks *exactly the same in UTF-8 as it did in ASCII*
- Others include UTF-7, UCS-4

Encodings:	H	E	L	L	O	
UTF-16:	00 48	00 65	00 6C	00 6C	00 6F	(two bytes/char)
UTF-8:	48	65	6C	6C	6F	
	H	E	L	L	O	

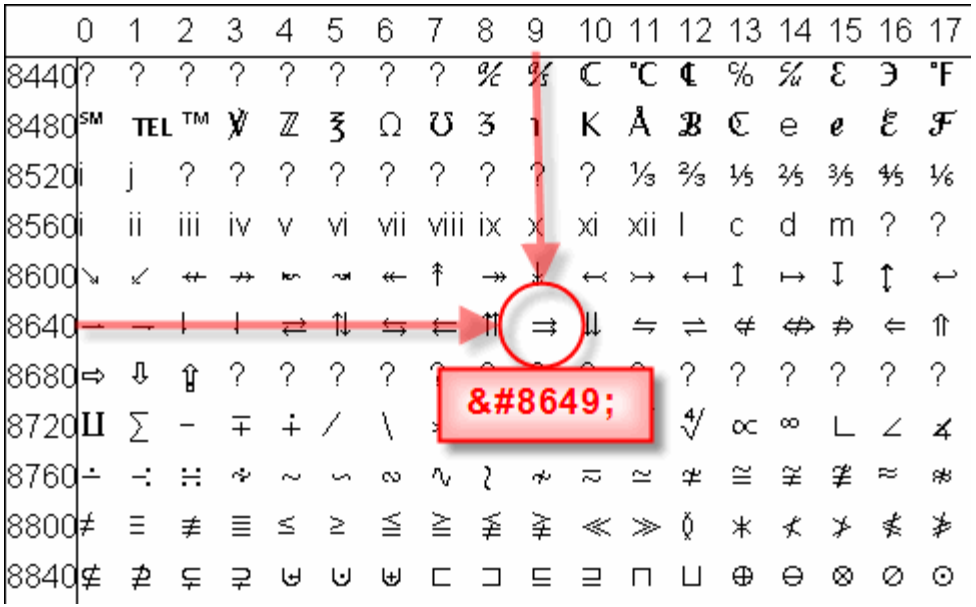
The Windows charmap utility






```
                 中    国    是    美    丽
0000000 feff 4e2d 56fd 662f 7f8e 4e3d
0000016 7684 56fd 5bb6 3002
                 的    国    家    。
```

Unicode entities in HTML



Someone gives you a document. How do you know the encoding?

Email:

Content-Type: text/plain; charset="UTF-8"

HTML:

- `<html><head>`
  `<meta http-equiv="Content-Type" content="text/html; charset=utf-8">`
- Often missing, so clever browsers will try to figure out the language and encoding from the frequency distribution of byte patterns

Python scripts: First line special comment, e.g.

```
# -*- coding: utf-8 -*-
```

or

```
# -*- coding: latin-1 -*-
```

Python 2.x support for Unicode

- In Python 2.5+, the default encoding for scripts is ASCII. (In Python 3, default is Unicode.)
- Unicode strings prefixed with ‘u’ or ‘U’ :

```
foo = u'abcdefghijk'
```

- Unicode string constructor:

```
s = unicode('foo')  # foo is an 8-bit string
```

- Unicode strings have same 8-bit string methods for searching, formatting, e.g. `s.find('bird')`
- Specific code points are written using the **\u escape sequence**, which is followed by four hex digits giving the code point.

```
>>> s = u'\u4e2d\u56fd'
>>> print s
中国
```

Source: http://docs.python.org/2/howto/unicode.html

Converting from Unicode to 8-bit strings in Python 2.x

- The `.encode` method of a Unicode string converts it to an 8-bit string in the requested encoding

```
>>> s = u'\u4e2d\u56fd'
>>> s.encode('utf-8')
'\xe4\xb8\xad\xe5\x9b\xbd'
>>> print s.encode('utf-8')
中国
>>> s.encode('ascii') # what now?
Traceback (most recent call last):
    ...
UnicodeEncodeError: 'ascii' codec can't encode
character u'\u4e2d' in position 0: ordinal not in
range(128)
```

Reference: http://docs.python.org/2/howto/unicode.html

Converting from 8-bit strings to Unicode strings

- Python 8-bit strings have a `.decode` method that converts to Unicode given the original encoding

```
>>> s = u'\u4e2d\u56fd'   # original 16-bit unicode string
>>> utf8_version = s.encode('utf-8')
>>> type(utf8_version), utf8_version
(<type 'str'>, '\xe4\xb8\xad\xe5\x9b\xbd')
>>> u2 = utf8_version.decode('utf-8')   # Convert UTF-8 to Unicode
>>> u == u2     # The two strings match
True
```

Regular expressions: Basics

You may have seen some kinds of regular expressions before

```
C:\>dir
 Volume in drive C is Windows
 Volume Serial Number is 5CCA-0D59

 Directory of C:\

07/21/2013  12:45 AM    <DIR>          cygwin
08/08/2013  08:15 PM    <DIR>          Perl64
09/12/2013  01:24 PM    <DIR>          Program Files
09/11/2013  04:48 PM    <DIR>          Program Files (x86)
09/09/2013  02:06 PM    <DIR>          Python27
09/13/2013  01:23 AM    <DIR>          temp
07/20/2013  04:59 PM    <DIR>          Users
09/15/2013  03:00 PM    <DIR>          Windows
               0 File(s)              0 bytes
               8 Dir(s)  135,135,703,040 bytes free

C:\>dir P*
 Volume in drive C is Windows
 Volume Serial Number is 5CCA-0D59

 Directory of C:\

08/08/2013  08:15 PM    <DIR>          Perl64
09/12/2013  01:24 PM    <DIR>          Program Files
09/11/2013  04:48 PM    <DIR>          Program Files (x86)
09/09/2013  02:06 PM    <DIR>          Python27
               0 File(s)              0 bytes
               4 Dir(s)  135,135,694,848 bytes free

C:\>dir *s
 Volume in drive C is Windows
 Volume Serial Number is 5CCA-0D59

 Directory of C:\

09/12/2013  01:24 PM    <DIR>          Program Files
07/20/2013  04:59 PM    <DIR>          Users
09/15/2013  03:00 PM    <DIR>          Windows
               0 File(s)              0 bytes
               3 Dir(s)  135,135,694,848 bytes free
```

Regular Expressions

- A concise and flexible means to "match" (specify and recognize) strings of text, such as particular characters, words, or patterns of characters.
- Similar regular expression syntax appears in many other tools
  - grep, flex, editors, ….
  - So you'll be able to re-apply most of what you learn here to other settings
- Light bedtime reading:
  - Unicode standard regular expression guidelines
  - http://unicode.org/reports/tr18/

Python raw string notation: `r'text'`

- Keeps regular expressions sane
- Without it, every backslash `'\'` in a regexp would need `'\'` prefix
- `r'\n'` is a two-character string containing `'\'` and `'n'`
- `'\n'` is a one-character string containing newline character
- Use `r'\\'` instead of `'\\\\'`
- Can be combined with Unicode 'u' prefix

```
>>> b = ur'\n'
>>> b
u'\\n'
>>> b.encode('utf-8')
'\\n'
>>> b.encode('utf-16')
'\xff \xfe \\ \x00n \x00'   [ff fe \ 00 n 00]
```

The Python `re` module

- Three Python functions:

  `re.match()` checks for a match only at beginning of string

  `re.search()` finds first occurrence of a pattern anywhere in string

  `re.findall()` finds all occurrences of a pattern, not just first one

- Some other regexp-enabled text operations:

  `re.split()`

  ```
  entries = re.split("\n+", text)
  ```

  `re.sub()`

  ```
  re.sub(r"(\w)(\w+)(\w)", repl, text)
  ```

`help(re.match)`

`match(pattern, string, flags=0)`

Try to apply the pattern at the <u>start</u> of the string, returning a match object, or None if no match was found.

Flag will be covered later.

`help(re.search)`

`search(pattern, string, flags=0)`

Scan through string looking for the first match to the pattern <u>anywhere</u> in the string, returning a match object, or None if no match was found.

A simple example

```
import re

str = 'a simple example!'

# want to see if ‘simple’ appears in the
# test string
match = re.search(r'simple', str)

if match:
    print 'found', match.group()
else:
    print 'did not find'
```

Basic Patterns

- Ordinary characters just match themselves.

  `match = re.search(r'dog','The lazy dog went to sleep.')`

  will match 'dog' in the right-hand string.

- Special characters:

  `\t`, `\r` tab, newline, return

  The meta-characters which do not match themselves because they have special meanings are:

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

What fancier patterns can we put in here?

https://developers.google.com/edu/python/regular-expressions

Very important single-character regular expression symbols

^ Beginning of the line `'^From: '`

Yes: `From: Kevyn` **No**: `It said, 'From:…`

$ End of the line (just before newline) `'Michigan$'`

Yes: `Michigan`**\n** **No**: `Michigan, U.S.A.`**\n**

. Matches any char except newline `\n` `'F..m:'`

Yes: `Farm:` **Yes**: `Foom:` **No**: `Firm.`

\s matches whitespace `'Pine`**\s**`apple'`

Yes: `Pine apple` **No**: `Pinesapple`

\S matches non-whitespace `'Pine`**\S**`pple'`

Yes: `Pineapple` **No**: `Pine pple`

Escape character

What if we really want to look for `'$'`?

Use an escape character: BACKSLASH

<u>Examples</u>:

`'\$19\.99'` will match `$19.99`

`'\\folder'` will match `\folder`

More useful special commands

`\d` Decimal digit, 0-9

`\D` Matches any non-digit character.

`\w` Matches a 'word' character: a **letter** or **digit** or underscore.
Note that although "word" is the mnemonic for this, it only matches a single word char, not a whole word.

`\W` Matches any non-word character.

`\b` Matches boundary between word \w and non-word \W chars:
`r'py\b' matches 'py', 'py.', or 'py!'`
but not `'python', 'py3', 'py2'`

`\B` Matches NOT at beginning or end of a word.
`r'py\B' matches 'python', 'py3', 'py2'`
but not `'py', 'py.', or 'py!'`

https://developers.google.com/edu/python/regular-expressions

Wildcards and matching repetitions

`*`	**Zero or more** of the previous thing
`+`	**One or more** of the previous thing
`?`	**Zero or one** of the previous thing
`{3}`	Matches exactly 3 of the previous thing
`{3,6}`	Matches between 3 and 6 of the previous thing
`{3,}`	Matches 3 or more of the previous thing

Wildcard examples

`ab*` will match ‘a’, ‘ab’, or ‘a’ followed by any number of ‘b’s.

`ab+` will match ‘a’ followed by at least one ‘b’; It will not match just ‘a’.

`ab?` will match either ‘a’ or ‘ab’.

Sets, ranges and alternatives

Set of characters

- `[aeiou]` Matches a single character in the given set {a, e, i, o, u}
- `[^aeiou]` Matches a single character NOT in the given set {a, e, i, o, u}

Example:

What substrings does `[aeiou]{2,}` match in

```
The eerie wind said "Oooo" and "Rrr".
```

```
The eerie wind said "Oooo" and "Rrr".
```

Why not O?

How you would use this in Python with findall

```
>>> import re
>>> s = "The eerie wind said Oooo and Rrrr"
>>> match = re.findall(r'[aeiou]{2,}', s)
>>> match
['ee', 'ie', 'ai', 'ooo']
```

Ranges of characters can be defined using […] and combined with sets

- <u>Valid</u>:

  `[A-Z]` Upper Case Roman Alphabet

  `[a-z]` Lower Case Roman Alphabet

  `[A-Za-z]` Upper/Lower Case

  `[A-F]` Upper Case (only A – F)

  `[0-9]` All Digits \d

  `[a-zA-Z0-9_]` \w

- <u>Invalid</u>:

  `[a-Z]`

  `[F-A]`

  `[9-0]`

Example using multiple operators

`^X-.*:\s[0-9.]+`

- What does this say?
  - We want strings that start (`'^'`) with `X-`
  - Followed by zero or more of any character `'.*'`
  - Then a colon (`':'`) and a whitespace `\s` char.
  - After the whitespace, look for **one or more** characters
    - That are either a digit (0-9) or a period
- Note that special characters are not active inside ranges, so `'.'` is treated as a period.

Example using multiple operators

`^X-.*:\s[0-9.]+`

Match? `xX-abd: 487.3` No

Match? `X-abd: 487.34.2` Yes

Match? `X-: .` Yes

Match? `X-abd: iii.3` No

Negation of Ranges of Regular Expressions

`[^0-9]`	Anything BUT digits
`[^a]`	Anything BUT a lower case a
`[^A-Z]`	Anything BUT upper case letters
`[^,]`	Anything BUT ,

What kind of strings does this match?

`^[^^]`

Match? `^foo` No

Match? `foo^` Yes

Strings that start with a character that is NOT `'^'`

Defining alternatives using the pipe | metacharacter

- `th(is|at|e other)`
  - matches 'this', 'that', or 'the other'
- `tha[nt]|re`
  - matches 'than' 'that' or 're'
- Each alternative can be a regular expression

`(success | failure code: [0-9]+ | maybe[!?]*)`

- Pipe is never greedy. As the target string is scanned:
  - REs separated by `'|'` are tried from left to right.
  - When one pattern completely matches, that branch is accepted.
  - This means that once A matches, B will not be tested further.
  - Even if it would produce a longer overall match.
- What does this match?

  `^(T|t)oday`

Group Extraction

<u>Problem</u>:

Often you want to extract parts of the matching text for later use. e.g. find email addresses, and extract user and hostname.

<u>Solution</u>: Use parentheses to create <u>groups</u> showing the parts you want to save for later.

```
str =  'My email addre Group 1 anta@ Group 2 edu. Hohoho.'
match = re.search(r'([\w.-]+)@([\w.-]+)', str)
if match:
  print match.group()    # the whole match
  print match.group(1)   # the username part
  print match.group(2)   # the hostname part
```

Help(re.findall)

```
findall(pattern, string, flags=0)
```

- Return a list of <u>all</u> non-overlapping matches in the string.
- If one or more groups are present in the pattern, return a list of groups.
- This will be a list of tuples if the pattern has more than one group.
- Empty matches are included in the result.

findall() Example

```
str = 'I have two email addresses: santa@umich.edu \
and santa@northpole.org. Hohoho.'

# Here re.findall() returns a list of all the found
# email strings
emails = re.findall(r'[\w\.-]+@[\w\.-]+', str)

['santa@umich.edu', 'santa@northpole.org']
```

findall() and Group Extraction

```
str = 'I have two email addresses: santa@umich.edu \
and santa@northpole.org. Hohoho.'

# Here re.findall() returns a list of all the found
# email strings
emails = re.findall(r'([\w\.-]+)@([\w\.-]+)', str)
email[0] = ('santa', 'umich.edu')
email[1] = ('santa', 'northpole.org')
```

Very useful power: You can refer back to an earlier group match within the <u>same</u> regular expression. How?

- \N where N is the group number
- \1 matches group 1 result

Example:

```
r'<(.*?)>(.*?)</\1>'
```

Matches tag pairs with <u>matching</u> begin/end tags

```
<X>foobar</X>    <ABCD>baz</ABCD>
```

Advanced matching: more subtle ways to modify *

- Greedy vs. non-greedy matching
- Zero-width lookahead

Options

- The option flag can be added as an extra argument to search(), findall() etc.,
  - e.g. `re.search(pat, str, re.IGNORECASE)`

- `re.IGNORECASE` Ignore upper/lowercase differences for matching, so 'a' matches both 'a' and 'A'.
- `re.DOTALL` Make the '.' special character match any character at all, including a newline; without this flag, '.' will match anything *except* a newline.
- `re.MULTILINE` Within a string made of many lines, allow ^ and $ to match the start and end of each line. Normally ^/$ would just match the start and end of the whole string.
- `re.UNICODE` Match against Unicode strings: invoke Unicode character properties for word-vs-nonword characters, etc.

Greedy Matching is the Default

- Python always tries to match as <u>much</u> as possible.
- Example:

```
str = 'the cat in the hat'
match = re.search(r'^(.*)(at)(.*)$', str)
```

Now, what do we have in

`match.group(1), match.group(2), match.group(3)?`

`'the cat in the h'`

`'at'`

`''`

Non-greedy Matching: Add an extra ? To your wildcard

- Non-greedy versions try to match as <u>minimally</u> as possible.

  `??` ,`*?` , `+?`, and `{}?`

- <u>Example 1</u>:

```
x = 'the cat in the hat';
match = re.search(r'^(.*?)(at)(.*)$', str)
```

Now, what do we have in

```
match.group(1), match.group(2) and
match.group(3)?
```

```
'the c'   'at'   ' in the hat'
```

- <u>Example 2</u>: `<H1>title</H1>`

  `<.*>` will match the whole string .

  `<.*?>` will match `<H1>`

Stop and look ahead: zero-width matching

- Problem:
  - We want to match any single character *q* that is not followed by *u*?
  - Why not use `q[^u]`

    Means: `q` followed by a character that is not a `u`

  `Iraqi population`

  `q[^u]` returns `qi` (q followed by i). This is two characters.
- What's the problem?
  - The regexp matcher has just 'used up' the `i` as part of this match and is now looking past it, at the 'space' character.
- But the 'i' may be important in an upcoming regexp match
  - Solution: check for the presence of 'not u' without letting regexp 'eat' it…
  - You do this by using a zero-width *negative lookahead assertion* **`q(?!u)`**
- Assertions do not 'use up' characters: they are zero-width, like start/end of line, or start/end of word
- This will match the single character *q only,* not trailing letters

Other types of zero-width assertions

- *Negative look<u>behind</u> assertion*:

  (?<!`abc`**)**`def` will <u>not</u> match `abcdef`, but will match `acbdef`

- *Positive look<u>behind</u> assertion* **(?<=**`abc`**)**`def` will first match `def`, then back up 3 characters and check for the contained pattern `abc`.

- What does `(?<=-)\w+` do?

  – Matches a word preceded by a hyphen

```
m = re.search('(?<=-)\w+','hard-boiled')
m.group(0): 'boiled'
```

What you should know

- How to write useful types of text matching patterns as regular expressions
- How to specify and extract groups in a match
- How to use the python `re` library functions to search and extract all matches in a text

Week 2 Review Resources

- Readings:
  - Severance, Chapter 11
- Excellent free online tool for debugging your tricky regexps:
  - http://gskinner.com/RegExr/
  - http://www.gskinner.com/RegExr/desktop/
- Other references:
  - https://developers.google.com/edu/python/regular-expressions
  - http://docs.python.org/2/howto/regex.html
  - http://docs.python.org/2/library/re.html

Next week sneak peak: XML, JSON, Web APIs


```
<?xml version='1.0' encoding='UTF-16'?>
<PublicFilings>
<Filing ID="306B3144-3E4F-48CF-98F1-C6BF455B6A6B" Year="2012" Received="2013-01-01T00:58:03.067"
Amount="15000" Type="FOURTH QUARTER REPORT" Period="4th Quarter (Oct 1 - Dec 31)"><Registrant xmlns=""
RegistrantID="6848" RegistrantName="Marshall Brachman" GeneralDescription="lobbyist" Address="634 A
Street, NE&#xD;&#xA;Washington, DC 20002" RegistrantCountry="USA" RegistrantPPBCountry="USA" /><Client
xmlns="" ClientName="ADAMS COUNTY COLORADO" GeneralDescription="" ClientID="12" SelfFiler="FALSE"
ContactFullname="MARSHALL BRACHMAN" IsStateOrLocalGov="TRUE" ClientCountry="USA" ClientPPBCountry="USA"
ClientState="" ClientPPBState="" /><Lobbyists><Lobbyist xmlns="" LobbyistName="BRACHMAN, MARSHALL A"
LobbyistCoveredGovPositionIndicator="NOT COVERED" OfficialPosition=""
/></Lobbyists><GovernmentEntities><GovernmentEntity xmlns="" GovEntityName="SENATE" /><GovernmentEntity
xmlns="" GovEntityName="Federal Aviation Administration (FAA)" /><GovernmentEntity xmlns=""
GovEntityName="HOUSE OF REPRESENTATIVES" /></GovernmentEntities><Issues><Issue xmlns=""
Code="BUDGET/APPROPRIATIONS" SpecificIssue="DERA funding regarding Rocky Mountain Arsenal&#xA;DoD
Appropriations for the above&#xA;TTHUD funding for railroad grade separation&#xA;funding for contract
tower program and commercial flight program" /><Issue xmlns="" Code="DEFENSE" SpecificIssue="DERA funding
regarding Rocky Mountain Arsenal&#xA;DoD Authorization" /><Issue xmlns="" Code="NATURAL RESOURCES"
SpecificIssue="land trade issues regarding the Rocky Mountain Arsenal NWP" /></Issues></Filing>
```


Lab 2: Regular expressions

Additional slides

Substitution

```
sub(pattern, repl, string, count=0,
flags=0)
```

- Return the string obtained by replacing the leftmost non-overlapping occurrences of the pattern in string by the replacement repl.
- repl can be either a string or a callable.
- If a string, backslash escapes in it are processed.
- If it is a callable, it's passed the match object and must return a replacement string to be used.

Substitution Example

```
str = 'My email is
santa@umich.edu. Hohoho.'
print re.sub(r'@[\w\.-]+',
'@northpole.org', str)
```

prints out My email is santa@northpole.org Hohoho.

Compile regex Patterns

- If a regex pattern is going to be reused, it is a good idea to compile it first.
- Example:

```
p = re.compile('\d+')
# search demo
m = p.search('12 drummers drumming, 11 pipers piping, 10
lords a-leaping')
if m:
  print 'Match found: ', m.group()
else:
  print 'No match'
# findall demo
print p.findall('12 drummers drumming, 11 pipers piping,
10 lords a-leaping')
```