

CS I

Introduction to Computer Programming

Lecture 19: November 20, 2012

Regular Expressions

$\text{th}.*\text{g}$

$^{\text{th}}.*\text{g\$}$

$\text{th}?nksg.*\text{g}$

$\text{th}[\text{an}]\{2\}.*\text{g}$

$\text{th}[\text{a-z}]^*\text{g}$

$\text{th}[^]^*\text{g}$



Last time

- Exception handling



Today

- Regular expressions
 - (advanced string processing)



Fun quote

- From Jamie Zawinski
 - (programmer for the Netscape Navigator web browser, which later became Firefox)
- "Some people, when confronted with a problem, think 'I know, I'll use regular expressions.' Now they have two problems."



What are regular expressions?

- Regular expressions are a very powerful way of processing strings
 - classifying strings
 - searching for patterns in strings
 - replacing parts of strings that match patterns



Applications of regular expressions

- Regular expressions are *very* heavily used in practical applications, for instance
 - bioinformatics
 - web programming
 - "data crunching"
 - and anywhere else strings are used



Disclaimers

- I can't cover everything about regular expressions in one lecture
 - Entire books have been written about them!
- I will focus on the most general/useful aspects of regular expressions as found in Python
 - see Python docs for more complete information
- Regular expressions are *not* Python-specific!
 - almost every language has a regular expression library, some more powerful than others



Terminology

- Regular expressions are also known as
 - `regexp`
 - `regexes`
 - `REs`
- for short



Theory

- There is a very rich theory behind regular expressions
- Regular expressions are related to
 - deterministic finite automata (DFA)
 - nondeterministic finite automata (NFA)
- Caltech's CS 21 course covers these topics
- You don't need to understand the theory to use regular expressions
 - but it's really interesting, so take CS 21 anyway!



In Python

- Regular expressions in Python require that you import the `re` module
- We will go over the main functions in this module later



The RE "language"

- Regular expressions are basically a kind of "little language" or "embedded language" for describing strings that exists inside Python
- This notion of "languages inside languages" may seem odd, but there are many examples of it in common use today



The RE "language"

- Other little languages:
 - **HTML** language for describing web pages
 - **XML** language(s) for describing arbitrary structured data
 - **JSON** language for describing key-value pairs
- Python modules exist for all of these little languages, so working with these languages in Python is pretty easy
- Today, we'll just talk about the RE language



Example 1

- We have a string representing a DNA sequence
- s = 'AGGTCGGAATGAGATCCTAAG...'**
- We want to find if a particular subsequence is found in the string **s**
 - for instance, the sequence '**ATTGCC**'
 - How do we solve this?



Example 1

- Several ways of doing this in Python
- You can write an explicit loop:

```
found = False
for i, e in enumerate(s):
    if s[i:i+6] == 'ATTGCC':
        found = True
        break
```



Example 1

- Several ways of doing this in Python
- You can use the `in` operator:

```
found = 'ATTGCC' in s
```



Example 1

- Several ways of doing this in Python
- Or you can use a *regular expression*:

```
import re # regular expression module
match = re.search('ATTGCC', s)
if not match:
    found = False
else:
    found = True
```

- (What **match** is will be explained shortly)



Example 1

- In this case, the regular expression is just the string to be matched ('ATTGCC')
- Using a regular expression here is no better than using the `in` operator
- However, if we make the problem a bit harder, we will see that regular expressions are much more convenient



Example 2

- New problem:
- Have an input DNA string, and want to look for substrings with this form:
 - A or T, A or T, A or T, G or C, G or C, G or C
 - e.g. **ATTGCC**, **AAAGGG**, **ATACCG**, etc.
- We can't use the **in** operator anymore!
 - At least, not without generating all of the $2^6 = 64$ possible substrings of that form
 - And even then it would be very inefficient



Example 2

- Let's try it with an explicit loop:

```
found = False  
  
for i, e in enumerate(s):  
    sub = s[i:i+6]  
    if sub[0] in 'AT' \  
        and sub[1] in 'AT' \  
        and sub[2] in 'AT' \  
        and sub[3] in 'CG' \  
        and sub[4] in 'CG' \  
        and sub[5] in 'CG':  
        found = True  
        break
```



Example 2

- Wow, was that ever tedious!
- How would you solve this with regular expressions?

```
import re
if not re.search(' [AT] [AT] [AT] [CG] [CG] [CG] ', s):
    found = False
else:
    found = True
```



Example 2

- We can do even better than this:

```
import re
if not re.search(' [AT]{3}[CG]{3}', s):
    found = False
else:
    found = True
```

- Moral: regular expressions can *drastically* simplify many string processing tasks



The regular expression "language"

- We've seen several things that we've called "regular expressions":
 - '**ATTGCC**'
 - ' [AT] [AT] [AT] [GC] [GC] [GC] '
 - ' [AT] {3} [GC] {3} '
- But what *are* regular expressions anyway?
- In Python, they are represented in two different ways



The regular expression "language"

1. Regular expressions are ordinary strings with a particular syntax
 - used by regular expression functions that expect these kinds of strings
2. Regular expressions are Python objects made from these strings
 - For now, we'll stick to the first form of regexps, and discuss the other later



The regular expression "language"

- A regular expression string is a string which represents a *set of matching strings*
- Idea: you can *match* the regular expression against any of the set of matching strings, but not against any non-matching string
- To understand this, we need to know the syntax of regular expression strings



Raw strings

- Normal Python strings have special characters encoded using backslash *escape sequences*
 - `\n` = newline, `\t` = tab, etc.
- This will make our lives difficult when writing regexp strings, because regexps assign their own meanings to characters
- To get around this, Python provides a variant of strings called a "raw string" which have no escape sequences



Raw strings

- A Python string preceded immediately by the letter **r** is a "raw string"
- A raw string is a string which consists exactly of all the characters in the string, with no escape sequences
- '**foo\nbar\n**' is a regular Python string
- **r'foo\nbar\n'** is a raw string
 - the '**\n**' part in the raw string is *not* a newline character, but just a two-character sequence (the **** character followed by the **n** character)



Raw strings

```
>>> print 'foo\nbar\n'
```

```
foo
```

```
bar
```

```
>>> print r'foo\nbar\n'
```

```
foo\nbar\n
```



Raw strings

- All regexps we describe will use raw strings to avoid problems
- In this lecture, when I write a string, I mean a raw string, so you should add the **r** prefix if you test the code out yourself
- Often, in practice, it doesn't matter, so in those cases you can use whatever you prefer (raw strings or regular Python strings)



Raw strings

- NOTE: Raw strings are *not* a special Python datatype!
- They are still just ordinary strings as far as Python is concerned
- The only difference is that the literal form of raw strings is different (*i.e.* you write the string in a different way)
- Similar to the situation with triple-quoted strings, docstrings, etc.



Regular expression syntax

- Regular expression strings assign new meanings to some of the characters in the string
- For instance, the characters:
 - . * + ? ^ \$ [] | () { }
 - all have special meanings inside regular expressions



Syntax: simple strings

- A string of non-special characters will match itself only
- Example: regexp string '**foo**' will match the string '**foo**' and nothing else
- If you want to match a string containing some of the special characters, you put a backslash (\) before the character
- Example: regexp string '**foo\?**' matches the string '**foo?**'



Syntax: the $*$ character

- The $*$ character means "zero or more of the preceding character"
 - actually it's more general than that, as we'll see shortly
- So the regexp ' a^* ' matches
 - '' (empty string)
 - 'a'
 - 'aa'
 - 'aaa'
 - etc.



Syntax: the ***** character

- We can incorporate the ***** character into other regexps
- The regexp '**fo*ba*r**' matches
 - '**fbr**'
 - '**fbar**'
 - '**foooooobr**'
 - '**foobaaaaaaaaar**'
 - etc.



Syntax: the + character

- The **+** character means "one or more of the preceding character"
- So the regexp '**f**o**tba+r**' matches
 - '**fobar**'
 - '**foooooobaar**'
 - '**foobaaaaaaaaar**'
 - etc.
- but not
 - '**fbr**'



Syntax: the ? character

- The **?** character means "zero or one of the preceding character"
- So the regexp '**f**o**?**b**a?**r' matches
 - '**f**br'
 - '**f**obr'
 - '**f**bar'
 - '**f**obar'
 - and that's it!



Syntax: the . character

- The . character means "any character except a newline"
- So the regexp '**f.b.r**' matches
 - '**fabar**'
 - '**fzbqr**'
 - '**f@b r**'
 - etc.



Combining special characters

- We can combine `.` and `*` or `+` to get useful regexps:
 - `'Mon.*on'` matches '**Monty Python**' and also '**Montreal Marathon**'
 - `'Par.+t'` matches '**Parrot**' and also '**Parrots are good to eat**'



Syntax: the () characters

- The (and) characters (parentheses) define a *group*
- A group is a sequence of characters that can be acted on as a whole by a special character
- For instance: '**(foo)+bar**' matches
 - 'foobar'
 - 'foofoobar'
 - 'foofoofoobar'
 - etc. (+ special character acts on the entire substring **foo**, not just on one character)



Syntax: the () characters

- What the *, +, ? etc. characters act on can be either
 - a single character
 - a group of characters surrounded by parentheses
- We will refer to these as a *pattern* from now on
 - So the *, +, ? characters act on the pattern that immediately precedes it



Syntax: the { } characters

- The { and } characters (curly braces) represent a number of repetitions
- {**n**} : previous pattern is repeated **n** times
- {**m**, **n**} : previous pattern is repeated between **m** and **n** times (at least **m**, at most **n**)
- where **m** and **n** are positive integers, **m <= n**



Syntax: the { } characters

- Example: '**(foo) {2}**' matches '**foofoo**' but not '**foo**' or '**foofoofoo**'
- Example: '**ab{3,5}c**' matches '**abbbc**', '**abbbb**', '**abbbbc**'



Syntax: the [] characters

- The [and] characters (square brackets) represent a *set of characters* that can exist in the string being matched
- The set matches if *any* of the characters in the set is present at that location in the string
- Inside the [and] characters there are some extra abbreviations
 - I never said this wasn't complicated! ☹



Syntax: the [] characters

- Example: the regexp '**[aeiou]**' matches any of the letters **a, e, i, o, u**
- Example: the regexp '**[ACGT] +**' matches any DNA string made of the letters **A, C, G, or T** in any order, with at least one character in the string



Syntax: the [] characters

- Shortcut 1: inside [] you can represent character ranges in order using the – character
 - **[A-Z]** = **[ABC...Z]**
 - **[A-Za-z]** = **[ABC...Zabc...z]**
 - **[0-9]** = **[012...9]**
- Can only do this in ascending order
- Can't do e.g. **[z-a]** to get **[zyx...a]**
 - (would mean the same thing as **[a-z]** anyway)



Syntax: the [] characters

- Shortcut 2: inside [and], if the first character is the ^ character, then match any character *except* for the characters inside the square brackets
- [^a] = any character except a
- [^0-9] = any character except a digit



Syntax: the | character

- The | (vertical bar) character represents alternatives; the regexp can match strings with *either* the pattern on the left or the pattern on the right of the | character
- Example: '**(foo)** | **(bar)**' will match either '**foo**' or '**bar**'
- Example '**(a+b)** | **(b+a)**' will match '**aaaab**', '**bbbba**', etc.



Syntax: the rest

- There are many more details and shortcuts in Python's regular expression syntax
- See the Python documentation for full details
- This will be enough for our purposes today



Interlude

- A classic Monty Python sketch



The `re` module

- Python's `re` module contains a number of functions for working with regular expressions
- We will discuss three of the most useful ones:
 - `re.match()`
 - `re.search()`
 - `re.sub()`
- but there are many more! (See the docs!)



re.match()

- The `re.match()` function takes two arguments:
 - a regexp, represented as a string
 - a string to match against
- and returns a "match object" if the string matches the regexp, or `None` if it doesn't
- The entire string doesn't have to match the regexp; only the *beginning* of the string needs to match for a match object to be returned



re.match()

- Examples:

```
>>> re.match('foo', 'foobar')
<_sre.SRE_Match object at 0x5adde8>
>>> re.match('foo', 'xxxxxx')
```

[None]

```
>>> re.match('foo', 'afoo')
```

[None]

- We will see what we can do with match objects shortly



re.match()

- Note that **None** is considered to be **False** when used in an **if** statement, so we can do this:

```
if not re.match('foo', 'xxxxx'):  
    found = False  
else:  
    found = True
```

- or just use **re.match(...)** as a boolean value itself:

```
found = re.match('foo', 'xxxxx')
```



re.match()

- More examples:

```
>>> re.match('a+b', 'aaaabbbbcccc')
<_sre.SRE_Match object at 0x5adde8>
>>> re.match('a+b', 'baaaaabbbbccc')
```

[None]

- However, often we want to be able to look for a regexp located at some unknown place *inside* a string, which leads us to...



`re.search()`

- `re.search()` is like `re.match()`, but it will return a match object if the regexp matches the string at *any* location in the string (not just starting at the beginning)
- If there are multiple matches, the first one is the one that's returned



re.search()

- Examples:

```
>>> re.search('foo', 'afoo')
```

```
<_sre.SRE_Match object at 0x5ade20>
```

```
>>> re.search('foo', 'xxx')
```

[None]

```
>>> re.search(' [AT]+', 'GGGCCAATTAACCC')
```

```
<_sre.SRE_Match object at 0x5ade20>
```



Match objects

- When a regular expression matches a string in functions like `re.match()`, `re.search()`, the function returns a match object
- These objects have several useful methods that give information about the match:
 - `start()`, `end()`, `span()`
 - `group()`
 - `groups()`



Match objects

- **start()**: returns the index of the first character matched
- **end()**: returns the index of one past the last character in the string
- **span()**: returns a tuple of (**start()**, **end()**)
- **group(n)**: returns the contents of the n^{th} group *i.e.* the part of the string that matched the n^{th} parenthesized part of the regexp
 - (group 0 is the entire match)
- **groups()**: returns the contents of all groups (except for group 0)



Match objects

- Example:

```
>>> m = re.search('def', 'abcdefghijklm')
>>> m.start()
3
>>> m.end()
6
>>> m.span()
(3, 6)
```



Match objects

- Example:

```
>>> m = re.match('([0-9]{3})-([0-9]{4})',  
'555-0101')  
  
>>> m.groups()  
(['555', '0101'])  
  
>>> m.group(0)  
'555-0101' # entire match  
  
>>> m.group(1)  
'555'  
  
>>> m.group(2)  
'0101'
```



Greed

- We say that regexp special characters like ***** and **+** are *greedy*
- They match as many characters/patterns as they possibly can
- This is important when searching for a regexp match in a larger string



Greed

- Example: we want to match the string '**Parrots are neat!**' using the regexp '**Par.*t**'
- There are two possible matches starting from the beginning of the string:
 - '**Parrot**'
 - '**Parrots are neat**'
- Which one should Python choose?



Greed

- In Python:

```
>>> s = 'Parrots are neat!'
```

```
>>> m = re.match('Par.*t', s)
```

```
>>> m.group(0)
```

```
'Parrots are neat'
```

- Python's regular expressions are *greedy*
- They match as much of the string as they possibly can
- Can cause problems if you're not aware of it!



Overcoming greed

- There are also *non-greedy* versions of some regexp operators:
 - `*?` is a non-greedy version of `*`
 - `+?` is a non-greedy version of `+`
- They look for the *shortest* match, not the longest

```
>>> s = 'Parrots are neat!'
>>> m = re.match('Par.*?t', s)
>>> m.group(0)
'Parrot'
```



Substitutions

- So far, we've seen how to match strings (or parts of strings) based on regexps
- Very often, you want to *find* a regexp and *substitute* something else for it
 - like the find/replace feature of editors
- With regexps, we use the **re . sub ()** function to do this



Substitutions

- `re.sub()` takes
 - a regexp
 - the string the regexp matches will be replaced with
 - the string to be substituted into
- and returns a new string, which includes the substitutions (substitutes *all* matches)
- Example:

```
>>> re.sub('MIT', 'Caltech',  
          'MIT is the best school.')
```

Caltech is the best school.



Substitutions

- `re.sub()`'s second argument (the replacement string) can also refer to groups matched in the first argument (the regexp)
- Notation: `\1` is the first group matched, `\2` is the second group, etc.
- Example: switching first and last names

```
>>> re.sub('([A-Za-z]+) ([A-Za-z]+)',  
         r'\2, \1', 'Joe Blow')
```

Blow, Joe



Substitutions

- Example: switching first and last names

```
>>> re.sub(' ([A-Za-z]+) ([A-Za-z]+)',  
         r'\2, \1', 'Joe Blow')  
  
Blow, Joe
```

- Note the raw string: `r'\2, \1'`; it's necessary!
 - otherwise it would have to be '`\\\2, \\\1`'
 - which is really tedious to write



Pattern objects

- We said before that Python regexps could be either strings or regexp objects
- We've worked with regexps as strings, but how do we create regexp objects?
- Answer: we use the `re.compile()` function
- This turns a regexp (represented as a string) into a regexp object
 - also known as a "pattern object"



Pattern objects

- Example:

```
>>> p = re.compile(' [ATGC]+')
```

- Now **p** is a pattern object:

```
>>> p
```

```
<_sre.SRE_Pattern object at 0x548f70>
```



Pattern objects

- You can use the pattern object to do matching, searching, and substituting using methods on the pattern object:

```
>>> m = p.match('AGGCCTTxxx')
>>> m.span()
(0, 7)

>>> m = p.search('xxxxAGCCGGTxxxx')
>>> m.span()
(4, 11)
```



Pattern objects

- You can use the pattern object to do matching, searching, and substituting using methods on the pattern object:

```
>>> p.sub(' ', 'xxxxAGCCGGTyyyy')  
xxxxyyyy
```

- This deletes the pattern from the string



Pattern objects

- Pattern objects don't let you do anything you couldn't do without them
- However, they are more efficient than using strings as regexps
 - Python can pre-process the regexp so that the regexp functions work more efficiently
- If you're using the same regexp over and over, you should probably `compile()` it into a pattern object



Final example

- We'll work through a more realistic example to see what real regexps are like
- We want to write a regexp that can match floating-point numbers with these properties:
 - optional minus sign at the beginning
 - decimal point is required
 - must have at least one digit before the decimal place
 - can have zero or more digits after the decimal place
 - optionally can have an exponent using **e** or **E** followed by the exponent (positive or negative)



Final example

- We will use groups to pick out
 - the minus sign (if any)
 - the part before the decimal point
 - the part after the decimal point (if any)
 - the exponent (if any)
- We will compile our regexp using `re.compile()` and use the `groups()` method on match objects to show the groups we matched



Floating-point number regexp

- The minus sign is optional, and will be a group
- The regexp we need for this is '`(-?)`'
- The digits before the decimal point are *not* optional
 - must have at least one digit
 - must be followed by a literal `.` character
 - these digits form a group, but the `.` is not part of the group
- You might think that the regexp for this part should be '`([0-9]+)`' – or is it?



Floating-point number regexp

- You might think that the regexp for this part should be '`([0-9]+)`.`'` – or is it?
- Recall: the `.` character matches any character except a newline
- For a literal `.` character, we have to "escape" it by preceding it with a backslash (`\.`)
- So the regexp for this part is '`([0-9]+)\.`'



Floating-point number regexp

- The part after the decimal place (but before the exponent) is zero or more digits
- It will also be a group
- This is easy to write as a regexp: **([0-9]*)**



Floating-point number regexp

- The exponent has three components:
 - the letter **e** or **E**
 - an optional sign (+ or -)
 - one or more digits
- of which the last two need to be in a group
- And: the entire exponent is optional!
- This requires a fairly complicated regexp:
([eE] ([+-]?[0-9]+)) ?



Floating-point number regexp

- Putting all of it together, we have:

```
'(-?) ([0-9]+) \. ([0-9]*) ([eE] ([+-] ? [0-9]+)) ?'
```

- We should make it a raw string, just to be on the safe side:

```
r'(-?) ([0-9]+) \. ([0-9]*) ([eE] ([+-] ? [0-9]+)) ?'
```

- We should compile it to make it into a regexp object (for efficiency):

```
>>> f = re.compile(r'(-?) ([0-9]+) \. ([0-9]*) ([eE] ([+-] ? [0-9]+)) ?')
```



Floating-point number regexp

- Let's test our shiny new regexp!

```
>>> f.match('1001.0').groups()  
('', '1001', '0', None, None)
```

- No sign, '1001' before the decimal point, '0' after, no exponent

```
>>> f.match('-1001.0e-10')  
('-', '1001', '0', 'e-10', '-10')
```

- A minus sign, '1001' before the decimal point, '0' after, and '-10' as the exponent
- It works! ☺



Summing up

- Regular expressions are complicated, and will take a while to get comfortable with
- However, they are also one of the most useful tools in a programmer's toolkit
- If you do a lot of string processing, you will use regular expressions more than you can probably believe at this point
- Whatever effort you put into learning them will be very richly rewarded



For reference

- The Python documentation:
 - <http://docs.python.org/library/re.html>
- Book: Mastering Regular Expressions by Jeffrey Friedl (O'Reilly and Associates)
 - the most definitive resource I know on regexps
 - (544 pages!)



Perl, .NET, Java, and More

2nd Edition



Mastering

Regular Expressions

O'REILLY®

Jeffrey E. F. Friedl

Next time

- More object-oriented programming
 - class "inheritance"
- Exceptions and classes
 - creating your own exceptions
 - raising exceptions explicitly
 - the **Exception** base class
- Also:



Happy Thanksgiving!

