1. **Last Time and Look Ahead**
   - **Context Managers**
     - Look Ahead

2. **Regular Expressions and Other Modules**
   - re
   - os
   - Queues
   - itertools
   - random
   - datetime
   - sys
The with statement

- `with expr as name:` begins a managed block
- Before the block is executed:
  - The `__enter__()` method of `expr` is called
  - The result is assigned to `name`
- The block is executed in a `try` block
- Any exceptions are passed to the `__exit__()` method of `expr`
- `__exit__(exc_type, exc_val, exc_trace_back)`
  - The arguments to `__exit__` can be used to handle certain errors
- `finally __exit__(None, None, None)` will be called

- `with EXPRES as VAR:
  BLOCK`

  Expands to ...
mgr = (EXPR)
exit = type(mgr).__exit__  # Not calling it yet
value = type(mgr).__enter__(mgr)
no_err = True
try:
    try:
        VAR = value  # Only if "as VAR" is present
        BLOCK
    except:
        no_err = False
        if not exit(mgr, *sys.exc_info()):
            raise
finally:
    if no_err:
        exit(mgr, None, None, None, None)
1. Last Time and Look Ahead
   - Context Managers
   - Look Ahead

2. Regular Expressions and Other Modules
   - re
   - os
   - Queues
   - itertools
   - random
   - datetime
   - sys
Lectures, Homeworks and Final Project

- Piazza Poll about lecture topics
- Four homeworks on special topics
- Rest of topics will not have an assignment
- Only final project to work on after Hw 10
- ~10 hrs of work per person
- Code will be due the last day of classes (Wed April 29)
- Demos during final exam period (Mon May 4 - Tues May 12)
Outline

1. Last Time and Look Ahead
   - Context Managers
   - Look Ahead

2. Regular Expressions and Other Modules
   - re
   - os
   - Queues
   - itertools
   - random
   - datetime
   - sys
What are Regular Expressions

- Formal specification of a language (set of strings)
- A regular language is one that can be recognized by a DFA
- Deciding if a string is in a regular language is very efficient
- Used for: Find/Replace, Syntax highlighting, Lexing (Compilers)
- Python (and most PLs) support more than regular languages
- Using non-regular features can lead to exponential run-times
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.

Python

Eric Kutschera (University of Pennsylvania)  CIS 192  February 20, 2015  9 / 32
Core Features 1

- Specify character literals to match directly
  - 'a' matches exactly 1 a
  - 'word' matches exactly the string word
- '*' matches 0 or more of the preceding expression
  - 'a*' matches: empty string, a, aa, aaa, ...
- '+' matches 1 or more
  - 'a+' matches: a, aa, aaa, ...
- '?' matches 0 or 1
  - 'a?' matches: empty string, a
- 'A|B' matches for either of two regexs A or B
  - 'foo|bar' matches: foo, bar
- 'AB' matches the regex A followed by regex B
  - 'a+b' matches: ab, aab, aaab, ...
Core Features 2

- `{m}` matches exactly m occurrences
- `{m, n}` matches between m and n occurrences
  - omitting m sets 0 as the lower bound
  - omitting n uses an infinite upper bound
- `'+*?` and `{m, n}` are greedy (biggest match)
  - `a+` will match aaa in aaab
- `'+?*???` and `{m, n}?` are lazy (smallest match)
  - `a+?` will match a in aaab
- `'^` matches the beginning of the string
  - `'^h` will find a match in `hi` but not in `blah`
- `$` matches the end of the string
- To match a literal special character uses a `\`
  - `\*` matches the literal `*`
Character Classes

- Specify a set of characters within ‘[]’
  - ‘[aeiou]’ matches any vowel
- Specify character ranges with ‘-’ inside ‘[]’
  - ‘[a-z]’ matches lowercase and ‘[0-9]’ matches digits
- A leading ‘^’ within ‘[]’ negates the set
  - ‘[^aeiou]’ matches any consonant
  - Actually r’[^aeiouAEIOU_\W\d]’ matches any consonant
- Built-in character classes
  - ‘\d’ → ‘[0-9]’
  - ‘\D’ → ‘[^0-9]’
  - ‘\s’ → r’[ \t\n\r\f\v]’
  - ‘\S’ → r’[^ \t\n\r\f\v]’
  - ‘\w’ → ‘[a-zA-Z0-9_]’
  - ‘\W’ → ‘[^a-zA-Z0-9_]’
Non-regular features 1

- ' (A)' matches the regex A and numbers it for reference
  - \( r'(\w)(\w)12' \) matches: abab, XyXy
- '(\?P<name>\w)' matches regex A and calls it name
  - '(\?P<delim>\w)\w*(\?P=delim)' matches: "stuff", x123x
- '(\?::\w)' matches the regex A but it can’t be referenced
  - \( r'(\?::(\w\w)\w+)' \) matches (): (one)(two)

Any named or numbered group can be accessed after the match

- '(\?## a comment)' is a comment and will be ignored
Look-aheads do not consume input but affect which strings match

' (?=A)' is a positive look-ahead for the regex A
- searching in 1<br/>2<br/>3<br/>4
- r'<br/>\d(?=<br/>)' matches <br/>2, <br/>3
- r'<br/>\d<br/>' will find only <br/>2<br/>

' (!?A)' is a negative look-ahead for the regex A

'turtle (?!soup)' matches 'turtle shell' not 'turtle soup'

' (?<=$A)' is a positive look-behind for the regex A

' (?<!A)' is a negative look-behind for the regex A

Look-behinds must match strings of fixed length

' (?(id/name)THEN|ELSE)' matches
- the regex THEN if group id/name exists
- the regex ELSE if group id/name does not exists
re.compile(regex) will create a regular expression object
  You can then use the methods of that object for matching

Methods of the re module can use a regex string directly
Compiling a legit `regular` expression can be exponential
  Using the compiled object afterwards will be linear
There are a bunch of flags that change the meaning of the regex
  Either specify them in the string with ‘(?aiLmsux)’
  Or in `re.compile(flags=)`
re.search(regex, string)
  returns the first match for regex in string
re.match(regex, string)
  returns a match if the regex matches at the start of string
re.fullmatch(regex, string)
  returns a match if the entire string matches regex
re.finditer(regex, string)
  returns an iterator of all non-overlapping matches
re.findall(regex, string)
  returns a list of all substrings that match in string
Basically [m.group(0) for m in re.finditer(...)]
re.sub(regex, repl, string)
  Extends string.replace(old, new) to regular expressions
re.split(regex, string)
  Extends string.split(sep) to regular expressions
Some of the re methods will return Match object, not strings

A Match object m supports:

- `m.group(id/name)`: Returns the string of the captured group
  - `m.group(0)`: returns the full string that matched
- `m.groupdict()`: A dictionary of named groups to strings
- `m.start(id/name)`: The index in the original string where the group (id/name) starts
- `m.end(id/name)`: the index at which the group ends
- `m.span(id/name)`: a tuple of start and end
Some people, when confronted with a problem, think
"I know, I’ll use regular expressions."
Now they have two problems. - Jamie Zawinski
Outline

1 Last Time and Look Ahead
   - Context Managers
   - Look Ahead

2 Regular Expressions and Other Modules
   - re
   - os
   - Queues
   - itertools
   - random
   - datetime
   - sys
Useful operations for doing os independent filepath manipulation

- Allow cross-platform code
- `os.path.join(path1, path2, path3, ...)`
  - Uses `/` on POSIX and `\` on Windows
- `os.path.realpath(path)` returns the absolute path of a file
- `os.path.exists(path)`
- `os.path.isfile(path)`
- `os.path.isdir(path)`
os Functions

- `os.getcwd()`: returns a sting for the current working directory
- `os.walk(top)`: A generator that for each directory in the tree rooted at top yields the tuple `(dirpath, dirnames, filenames)`
- `s = os.stat(path)`: a stat object with useful info on the file
  - `s.st_size` file size, `s.st_ctime` creation time, ...
- `os.mkdir(path)`: create a directory named path
- `os.chdir(path)`: cd into path
- `os.remove(path)`: rm path if not a directory
- `os.rmdir(path)`: rm `-r` path if it's an empty directory
- recursive deletion, copy, and move are in the `shutil` module
Outline

1. Last Time and Look Ahead
   - Context Managers
   - Look Ahead

2. Regular Expressions and Other Modules
   - re
   - os
   - Queues
   - itertools
   - random
   - datetime
   - sys
Queues

- **collections.deque**
  - append, extend **and** pop
  - appendleft, extendleft **and** popleft
  - fastest option for regular stack, queue, and deque

- **heapq**
  - min priority queue operations on built-in list objects
  - `heapify(seq)`: construction from list
  - `heappush(heap, x)`: push x into heap
  - `heappop(heap)`: pop smallest elem from heap

- **queue module**
  - Thread safe queues
  - slightly slower than deque and heapq
  - `queue.Queue`: FIFO
  - `queue.LifoQueue`
  - `queue.PriorityQueue`
1. Last Time and Look Ahead
   - Context Managers
   - Look Ahead

2. Regular Expressions and Other Modules
   - re
   - os
   - Queues
   - itertools
   - random
   - datetime
   - sys
**itertools Functions 1**

- `itertools.count(start=0, step=1)`: Generator for \([start, start + step, start + 2*step, ...]\)
- `itertools.repeat(x, times=None)`: Generator that continually yields \(x\) if \(times\) is None
  - Can specify a number of iterations with \(times\)
- `itertools.chain(iter1, iter2, ...)`: yields the objects of \(iter1\), then \(iter2\), then ...
- `itertools.islice(it, start, stop, step)`: Generator with the same intention as \(it[start:stop:step]\)
- `itertools.takewhile(pred, it)`
  - Generator for the elems of `it` up to the first elem where `pred(elem)` is False
- `itertools.dropwhile(pred, it)`
  - Everything after `itertools.takewhile(pred, it)`
- `itertools.permutations(it)`
  - Generator for all permutations of `it`
- `itertools.combinations(it, k)`
  - All size `k` subsets of elems from `it`
1 Last Time and Look Ahead
   - Context Managers
   - Look Ahead

2 Regular Expressions and Other Modules
   - re
   - os
   - Queues
   - itertools
   - random
   - datetime
   - sys
**random Functions**

- `random.seed()` initializes the random generator
  - Uses an os generated value by default
  - Can specify a specific seed to get repeatable numbers

- `random.random()` a float in \([0.0, 1.0)\)

- `random.uniform(a, b)` a float in \([a, b)\)

- `random.randrange(start, stop, step)`
  - An integer in `range(start, stop, step)`

- `random.choice(seq)`
  - An element of the sequence
  - `seq` must support `__len__` and `__getitem__`

- `random.shuffle(seq)` shuffles `seq` in place

- `random.sample(population, k)`
  - `k` unique elems from `population`
  - `population` can be a sequence or a set
1. Last Time and Look Ahead
   - Context Managers
   - Look Ahead

2. Regular Expressions and Other Modules
   - re
   - os
   - Queues
   - itertools
   - random
   - datetime
   - sys
datetime Objects

- Provides objects that have attributes for: day, year, month, minutes, ...
- Useful for uniformly representing dates
- Constructors for various formats
  - `datetime.strptime()` date strings (mm/dd/yyyy)
  - `datetime.fromtimestamp()` POSIX timestamps
- Can do comparisons with built-in operations (<, ==, ...)
- Most APIs support `datetime.datetime` objects
1. Last Time and Look Ahead
   - Context Managers
   - Look Ahead

2. Regular Expressions and Other Modules
   - re
   - os
   - Queues
   - itertools
   - random
   - datetime
   - sys
sys Functions

- `sys.argv` a list of command line arguments
  - `sys.argv[0]` is the name of the Python script
  - Use the `argparse` module for any non-trivial argument parsing
- `sys.stdin`, `sys.stdout`, `sys.stderr`
  - File handles that the interpreter uses for I/O