CIS192 Python Programming
Regular Expressions and Other Stuff

Harry Smith

University of Pennsylvania

February 22, 2018
Outline

1. Updates
   - What’s Next

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

- Vote for special topics!
- Let me know if there are others you’d like to suggest.
- ML, AI, DataViz, NLP, Web Apps … Further Functional Programming
Final Project

- Can work individually or with a partner
- ~10 hours of work per person
- Demos during CIS Project Fair
Outline

1 Updates
   • What’s Next

2 Regular Expressions and Other Modules
   • re
   • os
   • Queues
   • itertools
   • random
   • datetime
   • sys
String Matching

```python
>>> s = 'firefly'
>>> 'fly' in s
True
>>> s.find('fly')
4
>>> s.find('flies')
-1
>>> s.count('f')
2
>>> s.replace('fire', 'dragon')
'dragonfly'
```
Regular Expressions

- Compact way of specifying a set of strings that all have some property (like a substring)
- Can then check if a particular string belongs to the set
- i.e. does the string match the pattern?
Operators

- * – repeat 0 or more times
  - \(ab^*d\) matches \(ad, abd, abbd\) ...
  - \(a\ (bcd)^*d\) matches \(ad, abcdd, abcdabcdd\) ...

- + – repeat 1 or more times
  - \(ab+d\) matches \(abd, abbd\) ..., but not \(ad\)

- ? – repeat 0 or 1 times
  - \(ab?d\) matches \(ad\) and \(abd\)

- \(\{n,m\}\) – repeat between \(n\) and \(m\) times

- \(\{n\}\) – repeat exactly \(n\) times
Operators

- \( (. . . | . . . ) \) – means “or”
  - \((aa|bb)\) matches \(aa\) or \(bb\)

- \([. . . ]\) – also means “or”
  - \([abc]\) matches \(a\), \(b\), or \(c\)
  - can also write \([a-c]\)
  - equivalent to \((a|b|c)\)

- \([ ^. . . ]\) – means “not”
  - \([ ^5\]\) matches any character except 5
  - \([ ^0-9\]\) means not 0 or 1 or 2 ...
Other Special Characters

- . – matches any single character
  - .* matches any string
- ^ – matches the beginning of a string
  - note: not the same as [^ . . . ]!
- $ matches the end of a string

If you actually want to use any of the operators / characters mentioned above, need to escape them with a backslash.
Match Objects

```python
code:
>>> import re
>>> e = 'test'
>>> re.search(e, 'test1.txt')
<_sre.SRE_Match object at 0x107667b90>
>>> re.search(e, 'notes1.txt')

>>> re.search(e, 'othertest.pdf')
<_sre.SRE_Match object at 0x107667b90>
```
What if what we really want is a file whose name has the format: test(number).(txt or doc)?

```python
>>> e = 'test([0-9]+)\.(txt|doc)'
>>> re.search(e, 'test1.txt')
<_sre.SRE_Match object at 0x10768a918>
>>> re.search(e, 'test20.doc and more')
<_sre.SRE_Match object at 0x10768aa08>
>>> re.search(e, 'othertest.pdf')
>>>
Groups

```python
>>> e = 'test([0-9]+)\.(txt|doc)'
>>> m = re.search(e, 'test20.doc and more')
>>> m.group()
'test20.doc'
>>> m.groups()
('20', 'doc')
>>> m.group(1)
'20'
>>> m.group(2)
'doc'
```
>>> e = 'test(?P<number>[0-9]+)\.(?P<type>txt|doc)'
>>> m = re.search(e, 'test20.doc and more')
>>> m.group()
'test20.doc'
>>> m.groups()
('20', 'doc')
>>> m.group('number')
'20'
>>> m.group('type')
'doc'
Multiple Matches

`re.search` just returns one match object for the first match. Use `re.finditer` to return an *iterator* of all matched objects.

```python
>>> e = '[0-9]+'
>>> s = "I have 3 cats, 2 dogs, and 1 fish."
>>> iterator = re.finditer(e, s)
>>> for match in iterator:
...    print(match.group())
3
2
1
```
Multiple Matches

>>> e = '([A-Z]{3}) ([0-9]{3})'
>>> s = "I took CIS 110 and now I’m taking CIS 120."
>>> for match in re.finditer(e, s):
...    print(match.group(), match.groups())
...
CIS 110 ('CIS', '110')
CIS 120 ('CIS', '120')
Special Character Classes

- \d - any decimal digit ([0-9])
- \D - any non-decimal digit
- \s - any whitespace character
- \S - any non-whitespace character
- \w - any alphanumeric character ([a-zA-Z0-9_])
- \W - any non-alphanumeric character
- \b - word-boundary

To use them, need to use raw strings, e.g.:

- e = r’\b\w+\b’ – matches a single word
- e = r’([A-Z]{3}) (\d{3})’ – matches courses
Special Character Classes

```python
>>> e = r'\b\w+\b'
>>> s = "Hello! How are you? Bye."
>>> for match in re.finditer(e, s):
...    print(match.group())
...
Hello
How
are
you
Bye
```
Referring to Previous Groups

```python
def e = r'\(\d+\).*\1'
>>> s = "123 some text 123 more text."
>>> m = re.search(e, s)
>>> m.group()
'123 some text 123'
```
Referring to Previous Groups

Catch duplicate words:

```python
>>> e = r'\b\w+\s+\1'
>>> s = "This is the the the course."
>>> m = re.search(e, s)
>>> m.group()
'the the'
```
Referring to Named Groups

```python
>>> e = r'Hi (?P<name>\b\w+). Bye (?P=name).'
>>> re.search(e, 'Hi Sara. Bye Sara.')
<_sre.SRE_Match object at 0x1097f5a80>
>>> re.search(e, 'Hi Sara. Bye Susan.')
```

Harry Smith  (University of Pennsylvania)  CIS 192 Lecture 5  February 22, 2018  21 / 41
Compiling

`re.search(e, s)` is equivalent to:

```python
>>> e = re.compile(e)
>>> e.search(s)
```

You can save a compiled regex object for reuse!
Greedy Matching

Default matching is greedy:

```python
>>> s = '(1+4) + (2+5) + (3+6)'
>>> e = r'\(.*\)'
>>> m = re.search(e, s)
>>> m.group()
'(1+4) + (2+5) + (3+6)'
```
Greedy Matching

Use the quantifier \( ? \) to match non-greedily:

```python
greedy_match = (1+4) + (2+5) + (3+6)
expression = r'(.*?\)')
for match in re.finditer(expression, greedy_match):
    print(match.group())
```

... (1+4)
... (2+5)
... (3+6)
Replacing

Want to replace “CIS” with “Comp. Sci.”:

```python
>>> s = "She is taking CIS 120 and CIS 160."
>>> re.sub('CIS', 'Comp. Sci.', s)
'She is taking Comp. Sci. 120 and Comp. Sci. 160.'
```
Replacing

Want to replace parentheses with brackets:

```python
>>> s = '(1+4) + (2+5) + (3+6)'
>>> e1 = r'\((.*?))\'
>>> e2 = r'\[\1]\'
>>> re.sub(e1, e2, s)
'[1+4] + [2+5] + [3+6]'
```
Replacing

- The replace argument can be a function instead of a regex.
- The input to the function is a match object.
- The function gets applied to each match.

```python
>>> s = 'Hello World'
>>> e = '(\b\w+\b)'
>>> silly_func = lambda match: 'Wow!'
>>> re.sub(e, silly_func, s)
'Wow! Wow!'
```
Replacing

Say we want to limit all numbers to at most 10:

```python
>>> s = '1, 5, 10, 20, 100'
>>> e = r'\d+'
>>> fix = lambda m: '10' if int(m.group()) > 10 else m.group()
>>> re.sub(e, fix, s)
'1, 5, 10, 10, 10'
```
os Module

- `os.getcwd()` - get current directory
- `os.chdir(path)` - change directories
- `os.listdir(path)` - list directory contents
- `os.mkdir(path)` - create a directory
- `os.remove(path)` - remove a file
- `os.rename(src, dst)` - rename a file
- ... and many more
os Module

```python
>>> os.getcwd()
'/Users/lilidworkin/root'
>>> os.listdir('.
[', 'tests', ', ', 'notes', ', ', 'syllabus.txt']
```
Use `os.path.join` to create pathnames:

```python
>>> os.path.join('folder/', 'file.txt')
'folder/file.txt'

>>> os.path.join('folder', 'file.txt')
'folder/file.txt'
```

Allow cross-platform code – / on Unix/Mac, \ on Windows!
os.path.isfile and os.path.isdir are boolean functions:

```python
>>> os.listdir('root')
['tests', 'notes', 'syllabus.txt']

>>> [f for f in os.listdir('root')
...     if os.path.isdir(f)]
[]

>>> [f for f in os.listdir('root')
...     if os.path.isdir(os.path.join('root', f))]
['tests', 'notes']
```
Use `os.stat` to get file information:

```python
>>> info = os.stat('log.txt')
>>> info.st_size
23  # size in bytes
>>> info.st_mtime
1383968428.0  # time of last modification
```
Use `os.walk` to generate a directory tree:

```python
for (root, dirs, files) in os.walk('root'):
    ... print(root, dirs, files)
    ...

root ['tests', 'notes'] ['syllabus.txt']
root/tests [] ['test1.txt', 'test2.txt']
root/notes [] ['notes1.txt', 'notes2.txt']
```
What if we just want the test files?

```python
l = []
for (_, _, files) in os.walk('root'):
    l.extend([f for f in files if 'test' in f])

>>> l
['test1.txt', 'test2.txt']
```
Queues

- **collections.deque**
  - `append`, `extend` and `pop`
  - `appendleft`, `extendleft` and `popleft`
  - Best option for regular 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
  - What about max priority queues??
    - Invert values
    - `heappush(-5)` instead of `heappush(5)`
    - A more "proper" solution (StackOverflow)

- **queue module**
  - Thread safe queues: slightly slower than deque and heapq
  - `queue.Queue`
  - `queue.PriorityQueue`
**itertools Functions**

- `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 Functions

- `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`
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
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
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