Opening URLs

- The module used for opening URLs is `urllib2`
- The method used is similar to the file `open` in syntax
- Returns a handler to the URL, which could be used as a handle to a file (`readlines`, `read` etc.)

```
>>> import urllib2
>>> r = urllib2.urlopen('http://python.org/')
>>> html = r.read(300)
>>> print html
<!DOCTYPE html PUBLIC "-//W3C//DTD XHTML 1.0
Transitional//EN" "http://www.w3.org/TR/xhtml1/
DTD/xhtml1-transitional.dtd">


<html xmlns="http://www.w3.org/1999/xhtml" xml:
lang="en" lang="en">

<head>
  <meta http-equiv="content-type"
content="text/html; charset=utf-8" />
  <title>Python Programming Language --
Official Website</title>
```

General Way

- Not all urls can be opened this way.
- There could be complicated operations such as communicating with he cgi-bin of the server; or some ftp server; etc.
- For that purpose, there are Requests and Opener objects
  - Requests can send along extra data to the server
  - Opener can be used for complicated operations.

```
>>> from urllib2 import Request
>>> req = Request('http://www.google.com/')
>>> brwser = 'Mozilla/4.0 (compatible; MSIE 6.0;
                  Windows NT 5.0)'
>>> req.add_header('User-Agent', brwser)
>>> opener = urllib2.build_opener()
>>> opened = opener.open(req)
>>> print opened.read(150)
<!doctype html><head><meta http-equiv=content-typ
content="text/html; charset=UTF-8"><title>hallo -
Google Search</title><script>window.google={kEI:"
>>>
```

On Error?

- In case of errors, one can use the exception to show the error messages
- Two Exceptions which come handy are `HTTPError` and `URLError`
- They have to be used in the same order when you write the code. Because HTTPError is a subclass of URLError
- See the example below.

```
1
2  from urllib2 import Request, urlopen, URLError, H
3  req = Request(someurl)
4  try:
5      response = urlopen(req)
6  except HTTPError, e:
7      print 'The server didn't fulfill the req.'
8      print 'Error code: ', e.code
9  except URLError, e:
10     print 'We failed to reach a server.'
11     print 'Reason: ', e.reason
12 else:
13     print 'everything is fine'
```

Regular Expressions - A recap

- What are they?
  A means to find out string patters, To match strings, To find substrings and so forth
- When not to use them?
  When they are unavoidable. In normal cases where one needs to check whether a string is a substring of another, then is could be easier and more understandable and perhaps more efficient to use the normal string methods.
- When to use them?
  When you know they must be.

Regular Expressions in Theory

- Finite Automata - NFA and DFA, Alphabets
- Books on Compilers give a good account of these
- Limitations : $(a^n b^n)$, palindromes

Meta Characters

- If you want to search for `''test''`, then easy.
- What if you don't know what you want to search for. For example a telephone number? (Which you don't know)
- There are some classes of characters which are dedicated to make the using of regular expressions possible.
- Normal characters match for themselves. E.g. `t` matches *t*.
- Some special characters don't match themselves.

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

[and] : These can be used to specify a class of characters.

[a-z] : stands for all the lowercase characters. The literal '-' has special meaning inside the square brackets.

[abc$] stands for the characters 'a', 'b', 'c' and the dollar sign.

Even though $ has special meaning in RE context, but inside [and]

^ : For negation/complementing a set

[^a-z] means everything which is not lowercase.

\ is perhaps the most important metacharacter.

It is used when a meta-character is to be matched.

\d : Every decimal digit. [0-9]

\D : Everything non-digit; [^0-9]

\s : Any whitespace; [\t\n\r\f\b]

\S : Any nonwhitespace character

\w : Any alpha-numeric; [a-zA-Z0-9_]

\W : Any non-alpha-numeric-character

Importance of DOT

The character "." matches everything but a newline.
Even that can be done using a different mode of the RE module, using `re.DOTALL`

Repeating Things

- * : `ca*t` would match ct, cat, caat, caaaat, ...
- + : `ca+t` would match all of them except for ct
- ? : `ca?t` would match only ct or cat
- $\{m,n\}$: Minimum m times, maximum n times.
  $ca\{2,4\}t$ would match caat, caaat and caaaat. But not anything else.

Repeating Things

- It is easy to see that * is nothing but $\{0,\}$
- Similarly, + is nothing but $\{1,\}$ and
- ? is $\{0,1\}$

a|b matches a or b.

^, $ match the beginning
and ending of a line.

\A, \Z match the beginning
and end of a string

'\A[abc]*\Z' matches all strings
which are combinations of a, b and c

\b matches word boundaries:
'class\b' match 'class next Thursday'
'class\b' doesn't match 'classified'

```
a[bcd]*b against 'abcbd'

a       The a in the RE matches.

abcbd   The engine matches [bcd]*,
        going as far as it can,
        which is to the end of the
        string.

Failure The engine tries to match b,
        but the current position is
        at the end of the string,
        so it fails.
```

```
abcb    Back up, so that [bcd]*
        matches one less character.

Failure Try b again, but the current
        position is at the last
        character, which is a "d".

abc     Back up again, so that [bcd]*
        is only matching "bc".

abcb    Try b again. This time but the
        character at the current
        position is "b", so it succeeds.
```

Using Them

- Compile them
- Match them
  - `match()` : Determine if the re matches the string
  - `search()` : Scan and find the matches
  - `findall()` : Find all the matches
  - `finditer()` : Return and iterator
- Use them

```
>>> import re
>>> p = re.compile('[a-z]+')
>>> p
<_sre.SRE_Pattern object at 80c3c28>
>>> p.match("")
>>> print p.match("")
None
>>> m = p.match( 'tempo')
>>> print m
<_sre.SRE_Match object at 80c4f68>
```

Using Them

- `group()` : The string matched
- `start()` : Start of the string
- `end()` : The End of the string
- `span()` : A tuple with (start, end)

```
>>> m.group()
'tempo'
>>> m.start(), m.end()
(0, 5)
>>> m.span()
(0, 5)
>>> print p.match('::: message')
None
>>> m = p.search('::: message') ; print m
<re.MatchObject instance at 80c9650>
>>> m.group()
'message'
>>> m.span()
(4, 11)
```

```
p = re.compile( ... )
m = p.match( 'string goes here' )
if m:
    print 'Match found: ', m.group()
else:
    print 'No match'
```

```
>>> p = re.compile('\d+')
>>> p.findall('12 drummers drumming,
              11 pipers piping,
              10 lords a-leaping')
('12', '11', '10')
```

```
>>> iterator = p.finditer('12 drummers drumming,
                          11 ... 10 ...')
>>> iterator
<callable-iterator object at 0x401833ac>
>>> for match in iterator:
```

```
19 ...         print match.span()
20 ...
21 (0, 2)
22 (22, 24)
23 (29, 31)
```

Tree Deletion

```
1   def __delete(self, item):
2       if item < self.data:
3           if self.lchild:
4               self.lchild = self.lchild.delete(item)
5       elif item > self.data:
6           if self.rchild:
7               self.rchild = self.rchild.delete(item)
8       else:
9           if self.isLeaf()    :
10              return None
11          if self.singleDad():
12              return self.singleDad()
13          lLarge = self.leastLarger()
14          self.data, lLarge.data = lLarge.data, self.
15          self.rchild = self.rchild.delete(item)
16      return self
```

Flatten A List

```
1  def flatten(e):
2      if not e:
3          return ()
4      if not isinstance(e, list):
5          return (e)
6      return flatten(e(0)) + flatten(e(1:))
7  ------------------------------------------------
8  def flatten(vl):
9      fl = ()
10     for el in vl:
11         if isinstance(el, list):
12             for e in flatten(el):
13                 fl += (e)
14         else:
15             fl += (el)
16     return fl
```

Flatten A List

```
1  def reverserec(wh):
2          if not wh: return wh
3          return reverserec(wh[1:]) + wh[0]
4
5  def reverserecM(wh):
6          if len(wh) == 1: return wh
7          m = len(wh) / 2
8          return reverserecM(wh[m:])
9                  + reverserecM(wh[:m])
10
11 def reverseStrNotWord(str):
12         wh  = str.split()
13         whr = reverserecM(wh)
14         return ' '.join(whr)
```

Graphs

- Nodes
- Edges
- In Python

Crawler/Spider

- Open an URL
- Write a re.query
- Make a graph
- Nodes are pages