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.

```

from urllib2 import Request, urlopen, URLError, H
req = Request(someurl)
try:
    response = urlopen(req)
except HTTPError, e:
    print('The server didn't fulfill the req.')
    print('Error code: ', e.code)
except URLError, e:
    print('We failed to reach a server.')
    print('Reason: ', e.reason)
else:
    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]

ˆ : For negation/complementing a set

[ˆa-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.

ˆ, $ 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:
...         print(match.span())
...
(0, 2)
(22, 24)
(29, 31)
```

Generators

Generators

- Iterator creators (So to speak)
- Regular functions which return without returning.
- Uses `yield` statement
- Each call of `next` resumes from where it left off.
- State/Data values stored

```

def reverse(data):
    for index in range(len(data)-1, -1, -1):
        yield data(index)

>>> for char in reverse('golf'):
...     print(char)
...
f
l
o
g
```

Generators ...

- Generators are the equivalent of class based Iterators
- `__iter__` and `next` are created automatically
- Saving the vales makes it easier. No need to separate initialization/storage of index.
- Automatic raising of Exception on termination.

Simulating Generators

- Can be simulated with normal functions.
    1. Start with an empty list.
    2. Fill in the list instead of the `yield` statement
    3. Then return an iterator of the list
    4. Same result

```

def r(data):
    for index in range(len(data)-1, -1, -1):
        yield data(index)


def rS(data):
    _list = ()
    for index in range(len(data)-1, -1, -1):
        _list.append(data(index))
    return iter(_list)
```

```

>>> import gs
>>> for x in gs.r('this is cool'):
...         print(x)
...
l o o c   s i   s i h t
>>> for x in gs.rS('this is cool'):
...         print(x)
...
l o o c   s i   s i h t
>>>
>>>
```

Parsers in Python

- XML
- HTML

XML Parser

- SAX (Simple API for XML)
  - Reads the file as required
  - Special methods are called when tags are opened/closed
- DOM
  - Reads the whole file in a go
  - The whole structure is readily accessible for use.

SAX Parser

- `xml.sax.make_parser()` gives a generic parser object.
- The parser object is an instance of XMLReader. (It can read and output structured XML)
- A content handler has to be implemented for the XMLReader (example)
- Contenthandler is a class which is implemented for the specific needs

ContentHandler

- `startDocument()/endDocument()` are called from reading and processing the XML-Codes
- `startElement(name, attrs)` is called whenever a new tag is opened
    - `name` is the name of the tag
    - `attrs` contains the attributes part of the tag. It is an attribute object.

Contenthandler

- `endElement(name)` is called when a tag is closed.
- `characters(str)` gives the CDATA in the parameter to be used.
- There is no guarantee that all the data inside would be given in a single instance. One has to collect data if needed. (Example)

```python

from xml.sax.handler import ContentHandler
class CDATAPrinter(ContentHandler):
    def startElement(self, name, attrs):
        self.cdata=''
    def endElement(self, name):
        if len(self.cdata.strip()) > 0:
            print(name, ':', self.cdata.strip())
    def characters(self, str):
        self.cdata += str
```

```
<something>
        <string>HA HA HA </string>
        <number>12 34 43 </number>
        <nothing> nothing </nothing>
</something>
--------------------------------------------
>>> import boo
>>> import xml.sax
>>> parser = xml.sax.make_parser()
>>> parser.setContentHandler(boo.CDATAPrinter())
>>> parser.parse('cal.xml')
string  : HA HA HA
number  : 12 34 43
nothing  : nothing
something  : nothing
>>>
```

HTML Parsing

- HTML is sometimes XML
- HTML tags need not be closed always
- HTML tags can have attributes and some have always

HTML Parsing

- Similar to XML parsing
- There is an abstract class `HTMLParser` which needs to be implemented for own purposes
- It contains the following methods
  - `handle_starttag(tag, attrs)`
  - `handle_endtag(tag)`
  - `handle_startendtag(tag,attrs)`
  - `handle_data(data) (for characters(str))`

HTML Parsing

- The HTMLParser has its own ContentHandler. Just calling `HTMLParser()` gives an instance of the class.
- For parsing, one has to feed the html-text to the parser. `parser.feed(hstring)`
- As far as it can, it would ignore the errors in the string. Sometimes EOF reaches before the error-limit is reached.
- To read a URL, the following code would be useful. `parser.feed(urllib2.open(URL).read())`

```
from HTMLParser import HTMLParser
class MyHTMLParser(HTMLParser):
        def handle_starttag(self, tag, attrs):
                print("Breaking In: ", tag)
        def handle_endtag(self, tag):
                print("Getting Out: ", tag)
        def handle_startendtag(self, tag, attrs):
                print("Empty Tag??: ", tag)
```

```
>>> import myhtmlparser
>>> import urllib2
>>> parser = myhtmlparser.MyHTMLParser()
>>> parser.feed(urllib2.urlopen("http://www.bing.
Breaking In:   html
Breaking In:   head
Empty Tag??:   meta
Breaking In:   script
Getting Out:   script
Breaking In:   script
Getting Out:   script
...
...
...
Breaking In:   script
Getting Out:   script
Getting Out:   body
```

```
18 Getting Out:    html
```

Decorator Pattern

- In object-oriented programming, the decorator pattern is a design pattern that allows new/additional behaviour to be added to an existing class dynamically.
- In Python one cannot say that to be the same with the Decorator; even though one can achieve the same functionality with decorators in python.
- So, what are decorators IN Python?

Functions taking Functions

- Functions can have pointers to other functions as parameters.
- A function which can take another function as its parameter and can achieve something there by could be mainly classified as a decorator.
  See example.

```
>>> def ourdecorator2(foo):
...     def newfoo():
...         print("We are to call", foo.__name__)
...         return foo()
...     return newfoo
...
>>>
>>> foo = ourdecorator2(foo1)
>>>
>>> foo()
We are to call foo1
Hello World
>>>
```

```
>>>
>>> def ourdecorator(foo):
...     print("We are to call", foo.__name__)
...     return foo()
...
>>> def foo1():
...     print("Hello World")
...
>>>
>>> foo  = ourdecorator(foo1)
We are to call foo1
Hello World
>>>
>>>
```

Similar to Macros

- Decorators are similar to MACROS in other programming languages
- They are usually used to make a wrapper around functions
- And of course, classes too.

Usage, then Creation

- Function decorators are placed above the function with the key-character '@'

```
@thedecorator
def foo():
....
```

- The interpreter compiles `foo` and calls the decorator with that as argument.
- The result of that replaces the code for `foo`

How to implement decorator

- Could be Functions or Classes.
- The condition is that whatever the decorator returns, that should be callable.
- An object is callable, if the method `__call__` is implemented.

```
class theDecorator(object):
    def __init__(self, f):
        print("inside theDecorator.__init__()")
        f()
    def __call__(self):
        print("inside theDecorator.__call__()")


@theDecorator
def foobar():
    print("inside foobar()")

print("Finished decorating foobar()")

foobar()
```

```
>>> import decorators.py
inside theDecorator.__init__()
inside foobar()
Finished decorating foobar()
inside theDecorator.__call__()
```

Observation

- From the output, it is clear that the `init` is called when the decorator is used.
- So, usually, the call to the function is done only in the `call` function.
- Once a function is decorated, the behaviour totally changes. The call goes only to the decorated code. (line number 4 of the output)

```
class loggerdeco(object):

    def __init__(self, f):
        self.f = f

    def __call__(self):
        print("Entering", self.f.__name__)
        self.f()
        print("Exited", self.f.__name__)

@loggerdeco
def func1():
    print("inside func1()")

@loggerdeco
def func2():
    print("inside func2()")
```

```

func1()
func2()
Entering func1


inside func1()
Exited func1
Entering func2
inside func2()
Exited func2
```

Using Functions

- The same can be achieved using functions, instead of classes.
- The decorator functions usually enclose the decorated function in between the decoration.
- This is done inside a subfunction (equivalent of `__call__` and the pointer to the subfunction is returned.

```
def loggerdeco(f):
    def new_f():
        print("Entering", f.__name__)
        f()
        print("Exited", f.__name__)
    return new_f

@loggerdeco
def func1():
    print("inside func1()")

@loggerdeco
def func2():
    print("inside func2()")
```

```
func1()
func2()
print(func1.__name__)


Entering func1
inside func1()
Exited func1
Entering func2
inside func2()
Exited func2
new_f
```

Observation

- The name of the functions have been changed to `new_f`.
- This can be changed by reassigning `new_f.__name__ = f.__name__`

There are many cool uses of decorators. You can see more examples at `http://wiki.python.org/moin/PythonDecorat`

```
class memoized(object):
    def __init__(self, func):
        self.func = func
        self.cache = {}
    def __call__(self, *args):
        try:
            return self.cache(args)
        except KeyError:
            self.cache(args) = value = self.func(*args)
            return value
        except TypeError:
            return self.func(*args)
    def __repr__(self):
        return self.func.__doc__

@memoized
def fibonacci(n):
    "Return the nth fibonacci number."
    if n in (0, 1):
        return n
    return fibonacci(n-1) + fibonacci(n-2)

for i in xrange(1, 100, 9):
    print(fibonacci(i))
```

```
(sadanand@lxmayr10 @ ~) time python memorized.py
1
55
4181
317811
24157817
1836311903
139583862445
10610209857723
806515533049393
61305790721611591
4660046610375530309

real        0m0.014s
user        0m0.008s
sys         0m0.000s
(sadanand@lxmayr10 @ ~)
```

Static Variables and Methods

- A static variable in a class has always the same value, independent of the instances.
- Static variables are class variables, they belong to the class than to the instances
- They are accessed by the name of the Class, rather than the instance.

```
class myStatic:
      instances = 0
      def __init__(self):
            myStatic.instances += 1

      def howmany(self):
            return myStatic.instances

x = myStatic()
print(x.howmany())
y = myStatic()
print(y.howmany())
print(x.howmany())
---------------
print(1, 2, 2)
```

Static Methods

- They have the same return value independent of the class instance
- They don't have the `self` parameter
- For the same reason, they cannot access any of the `self.*` objects.
- The keyword is a decorator named `@staticmethod`

```
class myStatic:
    instances = 0

    def __init__(self):
        myStatic.instances += 1

    @staticmethod
    def howmany():
        return myStatic.instances


x = myStatic()
print(myStatic.howmany())
y = myStatic()
print(myStatic.howmany())
========================
prints 1, 2 as expected.
```

Classmethod

- `@classmethod` is perhaps a special thing for python.
- The methods decorated with this gets as the initial variable a class which is the original class (not the instance)
- That helps the function to act like a normal method of the class, by accepting all the attributes and treat them as static as well.

```
class myStatic:
        instances = 0
        def __init__(self):
                self.addinstance()

        @classmethod
        def howmany(cls):
                return cls.instances
        @classmethod
        def addinstance(cls):
                cls.instances += 1

x = myStatic()
print(myStatic.howmany())
y = myStatic()
print(myStatic.howmany())
=============================
prints 1, 2 as expected.
```

```
class myStatic:
    instances = 0

    def __init__(self):
        self.addinstance()

    @classmethod
    def howmany(cls):
        return cls.instances

    @classmethod
    def addinstance(cls):
        cls.instances += 1

    def nastything(self):
        print("trying to be nasty")
        self.instances = -1
```

```
x = myStatic()
x.nastything()
print(myStatic.howmany())
print(x.howmany())
y = myStatic()
x.nastything()
print(myStatic.howmany())
======================
trying to be nasty
1
1
trying to be nasty
2
```

Single Instance

- The static methods can be used to create a singleton object/pattern
- They are classes for which there is only one instance at any given time.
- They could be implemented using
    1. The class instance could lie in a static variable
    2. The method which gets the instance can be made static.

Anonymous Classes

- New classes could be defined inside functions and returned.
- Such are called anonymous classes
- Anonymous classes can also be created using `classobj`

```
def createclass(name):
        class myClass:
                def __init__ (self):
                        self.name = name
                def whoareyou(self):
                        print(self.name)

        return myClass

Creator = createclass('iAmCreator')
first = Creator()
first.whoareyou()
======================
prints iAmCreator as expected.
```

```
from new import classobj
class foo:
      def __init__(self):
            self.x = 'x'

foo2 = classobj('foo2', (foo,),
  {'bar': lambda self, x: 'got ' + str(x)})

print(foo2().bar(3))
print(foo2().x)
=================
prints got2, x as expected.
```

Meta Classe

- Not in the scope of our course.
- Creating tailormade classes / customized ones.
  `__metaclass__`

Some Philosophy?

If you'd like to know some python philosophy, then you may import the module `this`

The Zen of Python, by Tim Peters

Beautiful is better than ugly.
Explicit is better than implicit.
Simple is better than complex.
Complex is better than complicated.
Flat is better than nested.
Sparse is better than dense.
Readability counts.
Special cases aren't special enough to break the rules.
Although practicality beats purity.
Errors should never pass silently.
Unless explicitly silenced.
In the face of ambiguity, refuse the temptation to guess.
There should be one– and preferably only one –obvious way to do it.
Although that way may not be obvious at first unless

you're Dutch.
Now is better than never.
Although never is often better than *right* now.
If the implementation is hard to explain, it's a bad idea.
If the implementation is easy to explain, it may be a good idea.
Namespaces are one honking great idea – let's do more of those!

Tab Complete

- Getting Tab Complete (like Bash) in python prompt.

```

try:
    import readline
except ImportError:
    print("Unable to load readline module.")
else:
    import rlcompleter
    readline.parse_and_bind("tab: complete")
```

Problems

- Open an URL (from commandline)
- Use HTML parser to parse and filter the links / Write a re.query
- NeverEnding Iterator
- Single instance class
- Recursive function tracer.