Functions

We have already seen functions. But only the simplest forms. We can have functions

- With arguments having default values
- With keywords as arguments
- With multiple arguments.

```
>>> def myfoo(bar, foobar=True):
...     print bar,
...     if foobar:
...         print "ha ha ha!"
...
>>> myfoo("hello")
hello ha ha ha!
>>> myfoo("hello", foobar=False)
hello
>>>
```

Default values taken only once

- The default value of the parameter is initialised only once and it stays the same if not specifically called. Look at the following example.

```
>>> def add(this, tothat=()):
...     for e in this:
...         tothat.append(e+1)
...     return tothat
...
>>> add((23, 34))
(24, 35)
>>> add((23, 34))
(24, 35, 24, 35)
>>> add((23, 34))
(24, 35, 24, 35, 24, 35)
>>> add((23, 34))
(24, 35, 24, 35, 24, 35, 24, 35)
>>> add((23, 34), (1, 2))
(1, 2, 24, 35)
>>>
```

Multiple Arguments

- Functions with a *-ed argument can have multiple arguemnt.
- The arguments would be packed in a tuple
- The *-ed argument must follow the other typed of arguments.

```
>>> def mularg(i, j, *rest):
...     print i+j
...     for k in rest:
...         print k
...
>>> mularg(1, 2)
3
>>> mularg(1, 2, 4)
3
4
>>> mularg('hello', 'world',
            'this', 'is', 'cool!')
helloworld
this
is
cool!
>>>
```

Docstrings

- Strings surrounded by three quotes at the beginning of functions could be used for documentation purposes.
- These strings contain newlines in them.

```
>>> def simpledoc():
...     """This is a simple hello
...        world program - just to reveal
...        the beauty of docstrings"""
...     print "Hello World"
...
>>> simpledoc.__doc__
'This is a simple hello\n        world program - j
the beauty of docstrings'
>>> print simpledoc.__doc__
This is a simple hello
       world program - just to reveal
       the beauty of docstrings
>>> help(simpledoc)
...
```

With expression

- Files are to be always closed after use.
- A keyword named `with`
- Using `with` helps automatic closing of files after use.
- The object which is used with `with` must have the methods - `__enter__` and `__exit__` implemented

```
1
2
3
4 with open(filename) as f:
5     for line in f:
6         print line
```

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)

```
1
2  from xml.sax.handler import ContentHandler
3  class CDATAPrinter(ContentHandler):
4      def startElement(self, name, attrs):
5          self.cdata=''
6      def endElement(self, name):
7          if len(self.cdata.strip()) > 0:
8              print name, ':', self.cdata.strip()
9      def characters(self, str):
10         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_endttag(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())`

```
1
2 from HTMLParser import HTMLParser
3 class MyHTMLParser(HTMLParser):
4         def handle_starttag(self, tag, attrs):
5                 print "Breaking In: ", tag
6         def handle_endtag(self, tag):
7                 print "Getting Out: ", tag
8         def handle_startendtag(self, tag, attrs):
9                 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
Getting Out:  html
```

Shelves

- A shelve is a persistent dictionary object in python
- A dictionary in the secondary storage
- Could be opened and used as needed.
- `open` and `close` are the usual methods needed.

```
>>> import shelve
>>> d = shelve.open("myfile.shelf")
>>> d('lala') = 'booboo'
>>> d('kiki') = 'myamya'
>>> d
{'lala': 'booboo', 'kiki': 'myamya'}
>>> d('xx') = range(4)
>>> d
{'lala': 'booboo', 'xx': (0, 1, 2, 3), 'kiki': 'm
>>> d.close()
>>>
(sadanand@lxmayr10
myfile.shelf.bak  myfile.shelf.dat  myfile.shelf.
(sadanand@lxmayr10
>>> import shelve
>>> d = shelve.open("myfile.shelf")
>>> d
{'kiki': 'myamya', 'xx': (0, 1, 2, 3), 'lala': 'b
```

Relevance of a Word

- Searching and indexing is done based on the relevance of the word.
- The simplest method could be the frequency of occurrence.
  - That would lead to a problem that 'the', 'a', etc. would get more relevance.
- A better method: `tf-idf`

tf-idf

- `tf-idf` is a measure or a benchmark to find the relevance of each word on the basis of its occurrence and frequency in each file.
- It can be calculated as follows.
  - $n_{i,j}$ is the number of occurrences of word w_i in the document d_j
  - D is the number of documents
  - D_i is the number of documents in which the word w_i occurs.

$$tf_{i,j} = n_{i,j} / \sum_k n_{k,j} \quad (1)$$

$$idf_i = \ln \frac{D}{D_i} \quad (2)$$

$$tfidf_{i,j} = tf_{i,j} \times idf_i \quad (3)$$

What with it?

- A higher value of tf-idf implies that the word has a higher frequency of occurrence in the less number of files where it appear.
- The common words 'the' or 'a' would occur in every file and that makes the denominator in idf_i larger - thereby making the tf-idf value smaller.
- Every word has a tfidf value for each file.
- `http://en.wikipedia.org/wiki/Tf-idf`

Problems

- Implement an HTMLParser
- Use the parser to filter the text from the documents pointed by the nodes of the graph
- Create tf-idf values.

Looking Forward

- How many more lectures?
- What more to be done?