UNIVERSITY OF OSLO

Faculty of Mathematics and Natural Sciences

Exam in **INF3331**
Day of exam: **2013-12-05**
Exam hours: **4**

This examination paper consists of 12 pages including 7 pages of appendices.
Appendices: **2 (Regex syntax and HTMLparser documentation)**
Permitted materials: **None**

Make sure that your copy of this examination paper is complete before answering.

It is possible to answer the exam in either Norwegian or English. For all the questions it is important to specify any assumptions you make, in particular in cases where you feel the question text is unclear.

1: Python-syntax (7 points)

Find errors in the following Python code. Write a suggestion with correct Python syntax for each case. For instance, `a = 4b` should be corrected to `a = 4*b`.

A)	`if __name__ = '__main__':`
B)	`mylist = range(0:100:10)`
C)	`def f(x)` `return 2+x**2`
D)	`for i in range[10]:` `  sum += 5i`
E)	`if a == 1,` `  b = a:`
F)	`import np as numpy` `a = np:arange(10)`
G)	`plt.title(nice figure)`

2: Vectorization (5 points)

The following Python script uses Monte-Carlo simulation to estimate the probability of getting at least three 6-es when you throw 10 dice.

```
import random
N = 1000
ndice = 10
nsix = 3
M=0
for i in range(N):
     sixes = 0
     for j in range(ndice):
          r = random.randint(1, 6)
          if r == 6:
               sixes += 1
          if sixes >= nsix:
               M += 1
p = float(M)/N
print 'probability:', p
```

Write a vectorized version of the script using NumPy. The answer should be a complete Python-script, including all import-statements.

Hint: `numpy.random.random_integers(n,size=(k,j))` generates an array with dimensions *k*, *j*, where each element is a random number between 1 and *n* (including the limits).

3: Vectorization (3 points)

The following function computes the weighted sum of two vectors and returns the answer as a list. The arguments `x,y` may be lists, NumPy arrays or similar Python objects.

```
def vector_sum(x,y,a=1,b=1):
    sum = []
    for xi,yi in zip(x,y):
        sum.append(a*xi+b*yi)
    return sum
```

Write a function that performs the same operation, as efficient as possible. The arguments `x,y` should now be NumPy arrays, and the function sould return a NumPy array. Include necessary import statements.

4. Regular expressions (5 points)

A Django based web site has the following content in the file urls.py:

```
from django.conf.urls.defaults import *
urlpatterns = patterns('',
   (r'^articles/2003/$', 'news.views.special_case_2003'),
   (r'^articles/(\d{4})/$', 'news.views.year_archive'),
   (r'^articles/(\d{4})/(\d{2})/$', 'news.views.month_archive'),
   (r'^articles/(\d{4})/(\d{2})/(\d+)/$', 'news.views.article_detail'),
)
```

Recall that each URL-pattern in Django is a tuple with two elements. The first element is a regular expression, and the second element is the name of a view function that is called if the regular expression matches. The view-function is called with an HTTPrequest as first argument, and any groups in the regular expression as additional arguments.

We want two changes to the functionality:

1. Only proper dates should match; this means years between 1900 and 2099, months between 1 and 12, and days between 1 and 31.
2. The extpressions should accept dates on the form yyyy/mm/dd and yyyy-mm-dd. It should be optional to write the day and month using one or two digits. For instance, 2013/12/4 and 2013/12/04 should be equivalent.

Change or extend the list of URL-patterns to incorporate these changes.

Hint: Regex-syntax for "or" is | ("pipe"). For instance, the regular expression (a[bc]|x[yz]) matches either *a* followed by *b* or *c*, or *x* followed by *y* or *z*. See also the enclosed regex documentation.

5: Classes (5 points)

Explain what happens in the following Python code:

```
>>> class g:
  ...     cnt = 0
  ...     def __init__(self):
  ...         g.cnt +=1
  ...

>>> g0 = g()
>>> g1 = g()
>>> g0.cnt
2
>>> g1.cnt
2
>>> g0.__dict__
{}
>>> g1.cnt +=1
>>> g0.cnt
2
>>> g1.__dict__
{'cnt': 3}
```

6: Classes (10 points)

Implement a class for vectors in 3D. The class should support the following operations:

```
>>> from Vec3D import Vec3D
>>> u = Vec3D(1, 0, 0)  # (1,0,0) vector
>>> v = Vec3D(0, 1, 0)
>>> str(u)           # pretty print
'(1, 0, 0)'
>>> repr(u)         # u = eval(repr(u))
'Vec3D(1, 0, 0)'
>>> len(u)          # Eucledian norm
1.0
>>> u[1]            # subscripting
0.0
>>> v[2]=2.5        #subscripting with assignment
```

You do not need to consider the numerical efficiency of the operations. Remark that the len-operator returns the norm (length) of the vector. For a vector $u=(x,y,z)$ the norm $e(u)$ is defined as $e(u) = (x^2+y^2+z^2)^{1/2}$.

7: HTML parser (10 points)

As a new employee in Foomatic Inc. you took over an assignment from a previous employee. You are to complete a script that collects image files from a website, and stores them locally. The script you got looks as follows;

```
import HTMLParser
import urllib
import sys

urlString = "http://www.python.org"

def getImage(addr):
    u = urllib.urlopen(addr)
    data = u.read()

    splitPath = addr.split('/')
    fName = splitPath.pop()
    print fName

    f = open(fName, 'wb')
    f.write(data)
    f.close()

class parseImages(HTMLParser.HTMLParser):
    #find image tags and call getImage

lParser = parseImages()

u = urllib.urlopen(urlString)
print u.info()

lParser.feed(u.read())
lParser.close()
```

Write the contents of the class parseImages so that the script works as intended. Documentation for HTMLParser is attached.

Appendix 1: Regular expression syntax

Special characters

```
.       # any single character except a newline
^       # the beginning of the line or string
$       # the end of the line or string
*       # zero or more of the last character
+       # one or more of the last character
?       # zero or one of the last character
```

Brackets, sequences and examples

```
[A-Z]   # matches all upper case letters
[abc]   # matches either a or b or c
[^b]    # does not match b
[^a-z]  # does not match lower case letters
.*    # any sequence of characters (except newline)
[.*]  # the characters . and *
^no   # the string 'no' at the beginning of a line
[^no] # neither n nor o
A-Z   # the 3-character string 'A-Z' (A, minus, Z)
[A-Z] # one of the chars A, B, C, ..., X, Y, or Z
```

More special characters

```
\n     # a newline
\t     # a tab
\w     # any alphanumeric (word) character
       # the same as [a-zA-Z0-9_]
\W     # any non-word character
       # the same as [^a-zA-Z0-9_]
\d     # any digit, same as [0-9]
\D     # any non-digit, same as [^0-9]
\s     # any whitespace character: space,
       # tab, newline, etc
\S     # any non-whitespace character
\b     # a word boundary, outside [] only
\B     # no word boundary
\.     # a dot
\|     # vertical bar
\[     # an open square bracket
\)     # a closing parenthesis
\*     # an asterisk
\^     # a hat
\/     # a slash
\\     # a backslash
\{     # a curly brace
\?     # a question mark
```

19.1. `HTMLParser` — Simple HTML and XHTML parser

> **Note:** The `HTMLParser` module has been renamed to `html.parser` in Python 3. The *2to3* tool will automatically adapt imports when converting your sources to Python 3.

New in version 2.2.

Source code: Lib/HTMLParser.py

This module defines a class `HTMLParser` which serves as the basis for parsing text files formatted in HTML (HyperText Mark-up Language) and XHTML. Unlike the parser in `htmllib`, this parser is not based on the SGML parser in `sgmllib`.

class `HTMLParser.`**`HTMLParser`**

An `HTMLParser` instance is fed HTML data and calls handler methods when start tags, end tags, text, comments, and other markup elements are encountered. The user should subclass `HTMLParser` and override its methods to implement the desired behavior.

The `HTMLParser` class is instantiated without arguments.

Unlike the parser in `htmllib`, this parser does not check that end tags match start tags or call the end-tag handler for elements which are closed implicitly by closing an outer element.

An exception is defined as well:

exception `HTMLParser.`**`HTMLParseError`**

`HTMLParser` is able to handle broken markup, but in some cases it might raise this exception when it encounters an error while parsing. This exception provides three attributes: `msg` is a brief message explaining the error, `lineno` is the number of the line on which the broken construct was detected, and `offset` is the number of characters into the line at which the construct starts.

19.1.1. Example HTML Parser Application

As a basic example, below is a simple HTML parser that uses the `HTMLParser` class to print out start tags, end tags and data as they are encountered:

```
from HTMLParser import HTMLParser

# create a subclass and override the handler methods
class MyHTMLParser(HTMLParser):
    def handle_starttag(self, tag, attrs):
        print "Encountered a start tag:", tag
    def handle_endtag(self, tag):
        print "Encountered an end tag :", tag
    def handle_data(self, data):
```

```
        print "Encountered some data  :", data

# instantiate the parser and fed it some HTML
parser = MyHTMLParser()
parser.feed('<html><head><title>Test</title></head>'
            '<body><h1>Parse me!</h1></body></html>')
```

The output will then be:

```
Encountered a start tag: html
Encountered a start tag: head
Encountered a start tag: title
Encountered some data  : Test
Encountered an end tag : title
Encountered an end tag : head
Encountered a start tag: body
Encountered a start tag: h1
Encountered some data  : Parse me!
Encountered an end tag : h1
Encountered an end tag : body
Encountered an end tag : html
```

19.1.2. `HTMLParser` Methods

`HTMLParser` instances have the following methods:

HTMLParser.**feed**(*data*)
: Feed some text to the parser. It is processed insofar as it consists of complete elements; incomplete data is buffered until more data is fed or `close()` is called. *data* can be either `unicode` or `str`, but passing `unicode` is advised.

HTMLParser.**close**()
: Force processing of all buffered data as if it were followed by an end-of-file mark. This method may be redefined by a derived class to define additional processing at the end of the input, but the redefined version should always call the `HTMLParser` base class method `close()`.

HTMLParser.**reset**()
: Reset the instance. Loses all unprocessed data. This is called implicitly at instantiation time.

HTMLParser.**getpos**()
: Return current line number and offset.

HTMLParser.**get_starttag_text**()
: Return the text of the most recently opened start tag. This should not normally be needed for structured processing, but may be useful in dealing with HTML “as deployed” or for re-generating input with minimal changes (whitespace between attributes can be preserved, etc.).

The following methods are called when data or markup elements are encountered and they

are meant to be overridden in a subclass. The base class implementations do nothing (except for `handle_startendtag()`):

`HTMLParser.`**`handle_starttag`**(*tag*, *attrs*)
This method is called to handle the start of a tag (e.g. `<div id="main">`).

The *tag* argument is the name of the tag converted to lower case. The *attrs* argument is a list of `(name, value)` pairs containing the attributes found inside the tag's `<>` brackets. The *name* will be translated to lower case, and quotes in the *value* have been removed, and character and entity references have been replaced.

For instance, for the tag `<A HREF="http://www.cwi.nl/">`, this method would be called as `handle_starttag('a', [('href', 'http://www.cwi.nl/')])`.

Changed in version 2.6: All entity references from `htmlentitydefs` are now replaced in the attribute values.

`HTMLParser.`**`handle_endtag`**(*tag*)
This method is called to handle the end tag of an element (e.g. `</div>`).

The *tag* argument is the name of the tag converted to lower case.

`HTMLParser.`**`handle_startendtag`**(*tag*, *attrs*)
Similar to `handle_starttag()`, but called when the parser encounters an XHTML-style empty tag (`<img ... />`). This method may be overridden by subclasses which require this particular lexical information; the default implementation simply calls `handle_starttag()` and `handle_endtag()`.

`HTMLParser.`**`handle_data`**(*data*)
This method is called to process arbitrary data (e.g. text nodes and the content of `<script>...</script>` and `<style>...</style>`).

`HTMLParser.`**`handle_entityref`**(*name*)
This method is called to process a named character reference of the form `&name;` (e.g. `&gt;`), where *name* is a general entity reference (e.g. `'gt'`).

`HTMLParser.`**`handle_charref`**(*name*)
This method is called to process decimal and hexadecimal numeric character references of the form `&#NNN;` and `&#xNNN;`. For example, the decimal equivalent for `&gt;` is `&#62;`, whereas the hexadecimal is `&#x3E;`; in this case the method will receive `'62'` or `'x3E'`.

`HTMLParser.`**`handle_comment`**(*data*)
This method is called when a comment is encountered (e.g. `<!--comment-->`).

For example, the comment `<!-- comment -->` will cause this method to be called with the argument `' comment '`.

The content of Internet Explorer conditional comments (condcoms) will also be sent to

this method, so, for `<!--[if IE 9]>IE9-specific content<![endif]-->`, this method will receive `'[if IE 9]>IE-specific content<![endif]'`.

`HTMLParser.`**`handle_decl`**(*decl*)

This method is called to handle an HTML doctype declaration (e.g. `<!DOCTYPE html>`).

The *decl* parameter will be the entire contents of the declaration inside the `<!...>` markup (e.g. `'DOCTYPE html'`).

`HTMLParser.`**`handle_pi`**(*data*)

This method is called when a processing instruction is encountered. The *data* parameter will contain the entire processing instruction. For example, for the processing instruction `<?proc color='red'>`, this method would be called as `handle_pi("proc color='red'")`.

> **Note:** The `HTMLParser` class uses the SGML syntactic rules for processing instructions. An XHTML processing instruction using the trailing `'?'` will cause the `'?'` to be included in *data*.

`HTMLParser.`**`unknown_decl`**(*data*)

This method is called when an unrecognized declaration is read by the parser.

The *data* parameter will be the entire contents of the declaration inside the `<![...]>` markup. It is sometimes useful to be overridden by a derived class.

19.1.3. Examples

The following class implements a parser that will be used to illustrate more examples:

```
from HTMLParser import HTMLParser
from htmlentitydefs import name2codepoint

class MyHTMLParser(HTMLParser):
    def handle_starttag(self, tag, attrs):
        print "Start tag:", tag
        for attr in attrs:
            print "     attr:", attr
    def handle_endtag(self, tag):
        print "End tag  :", tag
    def handle_data(self, data):
        print "Data     :", data
    def handle_comment(self, data):
        print "Comment  :", data
    def handle_entityref(self, name):
        c = unichr(name2codepoint[name])
        print "Named ent:", c
    def handle_charref(self, name):
        if name.startswith('x'):
            c = unichr(int(name[1:], 16))
        else:
            c = unichr(int(name))
        print "Num ent  :", c
    def handle_decl(self, data):
        print "Decl     :", data
```

```
parser = MyHTMLParser()
```

Parsing a doctype:

```
>>> parser.feed('<!DOCTYPE HTML PUBLIC "-//W3C//DTD HTML 4.01//EN" '
...             '"http://www.w3.org/TR/html4/strict.dtd">')
Decl     : DOCTYPE HTML PUBLIC "-//W3C//DTD HTML 4.01//EN" "http://www.w3.org/TR/htm
```

Parsing an element with a few attributes and a title:

```
>>> parser.feed('<img src="python-logo.png" alt="The Python logo">')
Start tag: img
     attr: ('src', 'python-logo.png')
     attr: ('alt', 'The Python logo')
>>>
>>> parser.feed('<h1>Python</h1>')
Start tag: h1
Data     : Python
End tag  : h1
```

The content of `script` and `style` elements is returned as is, without further parsing:

```
>>> parser.feed('<style type="text/css">#python { color: green }</style>')
Start tag: style
     attr: ('type', 'text/css')
Data     : #python { color: green }
End tag  : style
>>>
>>> parser.feed('<script type="text/javascript">'
...             'alert("<strong>hello!</strong>");</script>')
Start tag: script
     attr: ('type', 'text/javascript')
Data     : alert("<strong>hello!</strong>");
End tag  : script
```

Parsing comments:

```
>>> parser.feed('<!-- a comment -->'
...             '<!--[if IE 9]>IE-specific content<![endif]-->')
Comment  :  a comment
Comment  : [if IE 9]>IE-specific content<![endif]
```

Parsing named and numeric character references and converting them to the correct char (note: these 3 references are all equivalent to `'>'`):

```
>>> parser.feed('&gt;&#62;&#x3E;')
Named ent: >
Num ent  : >
Num ent  : >
```

Feeding incomplete chunks to **`feed()`** works, but **`handle_data()`** might be called more than once:

```
>>> for chunk in ['<sp', 'an>buff', 'ered ', 'text</s', 'pan>']:
...     parser.feed(chunk)
```

```
...
Start tag: span
Data     : buff
Data     : ered
Data     : text
End tag  : span
```

Parsing invalid HTML (e.g. unquoted attributes) also works:

```
>>> parser.feed('<p><a class=link href=#main>tag soup</p ></a>')
Start tag: p
Start tag: a
     attr: ('class', 'link')
     attr: ('href', '#main')
Data     : tag soup
End tag  : p
End tag  : a
```