strutturati

November 23, 2017

1 Documenti strutturati (cap 15)

I documenti strutturati si possono rappresentare come alberi i cui nodi definiscono le differenti sezioni del documento, che a loro volta contengono testo formattato.

L'elaborazione di un documento strutturato richiede come primo passo il PARSING, cioè l'analisi della struttura sintattica del documento.

Il risultato del parsing è il cosiddetto albero di parsing che, visitato ricorsivamente, permette di ottenere informazioni o di modificare il documento in vari modi.

Vedremo come fare il parsing e poi l'elaborazione di documenti HTML, il formato delle pagine Web.

1.1 Per formattare il testo molti linguaggi usano dei marcatori (tag), l' HTML è uno di questi.

i tag servono a delimitare parti che hanno specifiche proprietà: paragrafi, immagini link, tabelle ecc.ecc.

Nell'albero che costruiremo ogni tag sarà un nodo e i tag al suo interno ne saranno figli.

ogni nodo ha quattro attributi.

TAG è il nome del tag, ad esempio 'p'.

ATTR è il dizionario degli attributi, che rimarrà vuoto se l'elemento non ha attributi altrimenti ogni chiave del dizionario è un attributo ed ha associato il suo valore.

CONTENT è il contenuto del nodo. Se il nodo è di testo (con tag *text*), il contenuto è la stringa di testo, altrimenti è una lista dei nodi figli, che può anche essere vuota.

CLOSED closed è True se il tag del nodo ha la chiusura, altrimenti è False

```
In [1]: import html
```

```
        class HTMLNode:
            def __init__(self, tag, attr, content, closed=True):
                self.tag = tag              # dizionario degli attributi
                self.attr = attr            # testo per nodi _text_ o lista dei figli
                self.content = content      # True se il nodo ha la chiusura
                self.closed = closed        # per distinguere i nodi testo

            def open_tag(self):
                '''Ritorna la stringa del tag di inizio.'''
                if self.tag=='_text_':
```

```
            return self.tag
        s = '<' + self.tag
        for k, v in self.attr.items(): # usiamo escape per i valori
            s += ' {}="{}"'.format(k, html.escape(v,True))
        s += '>'
        return s

    def close_tag(self):
        '''Ritorna la stringa del tag di fine.'''
        return '</'+ self.tag + '>'

    def to_string(self):
        '''Ritorna la stringa del documento HTML che corrisponde all'albero di questo no
        if self.tag=='_text_':
            return html.escape(self.content, False) # usiamo escape per i caratteri spec
        doc = self.open_tag()
        if self.closed:
            for figlio in self.content:
                doc += figlio.to_string()
            doc += self.close_tag()
        return doc

    def print_tree(self, level=0):
        '''Stampa l'albero mostrando la struttura tramite indentazione'''
        if self.tag=='_text_':
            print('  '*level + '_text_ ' + repr(self.content))
        else:
            print('  '*level + '<' + self.tag+ '>')
            #print('  '*level + str(self))
            for figlio in self.content:
                figlio.print_tree(level + 1)
```

Per costruire l'albero useremo il modulo html.parser della libreria standard ed in particolare la classe HTMLParser. Per creare il nostro parser dovremo modificare il comportamento della classe standard ridefinendone alcuni metodi.

Coda fa il parser quando incontra un tag di inizio?

Cosa deve fare il parser quando incontra un tag di fine?

Cosa deve fare il parser quando incontra del testo?

```
In [2]: from html.parser import HTMLParser

        class MyHTMLParser(HTMLParser):
            def handle_starttag(self, tag, attrs):
                print("trovato uno start tag:", tag)

            def handle_endtag(self, tag):
```

```
            print("trovato un  end tag :", tag)

        def handle_data(self, data):
            print("trovato del testo  :")
            #print("Encountered some data  :", data)
```

In [3]:
```
def fparse(fhtml):
    '''Esegue il parsing HTML del file fhtml e ritorna la radice dell'albero .'''
    with open(fhtml) as f: html=f.read()
    parser = MyHTMLParser()
    parser.feed(html)
```

In [4]:
```
fparse('semplice.html')
```

```
trovato del testo  :
trovato uno start tag: html
trovato del testo  :
trovato uno start tag: body
trovato del testo  :
trovato uno start tag: h1
trovato del testo  :
trovato un  end tag : h1
trovato del testo  :
trovato uno start tag: p
trovato del testo  :
trovato uno start tag: em
trovato del testo  :
trovato un  end tag : em
trovato del testo  :
trovato uno start tag: strong
trovato del testo  :
trovato un  end tag : strong
trovato del testo  :
trovato un  end tag : p
trovato del testo  :
trovato uno start tag: p
trovato del testo  :
trovato uno start tag: a
trovato del testo  :
trovato un  end tag : a
trovato del testo  :
trovato un  end tag : p
trovato del testo  :
trovato uno start tag: img
trovato del testo  :
trovato un  end tag : body
trovato del testo  :
trovato un  end tag : html
```

```
In [5]: import html.parser

        class _MyHTMLParser(html.parser.HTMLParser):
            def __init__(self):
                '''Crea un parser per la class HTMLNode'''
                super().__init__()
                self.root = None
                self.stack = []
            def handle_starttag(self, tag, attrs):
                '''Metodo invocato per tag aperti'''
                closed = tag not in ['img','br']
                node = HTMLNode(tag,dict(attrs),[],closed)
                if not self.root:
                    self.root = node
                if self.stack:
                    self.stack[-1].content.append(node)
                if closed:
                    self.stack.append(node)
            def handle_endtag(self, tag):
                '''Metodo invocato per tag chiusi'''
                if self.stack and self.stack[-1].tag == tag:
                    self.stack[-1].opentag = False
                    self.stack = self.stack[:-1]
            def handle_data(self, data):
                '''Metodo invocato per il testo'''
                if not self.stack: return
                self.stack[-1].content.append(HTMLNode('_text_',{},data))
            def handle_entityref(self, name):
                '''Metodo invocato per caratteri speciali'''
                if name in name2codepoint:
                    c = unichr(name2codepoint[name])
                else:
                    c = '&' + name
                if not self.stack: return
                self.stack[-1].content.append(HTMLNode('_text_',{},c))
            def handle_charref(self, name):
                '''Metodo invocato per caratteri speciali'''
                if name.startswith('x'):
                    c = unichr(int(name[1:], 16))
                else:
                    c = unichr(int(name))
                if not self.stack: return
                self.stack[-1].content.append(HTMLNode('_text_',{},c))
            def handle_comment(self, data):
                '''Metodo invocato per commenti HTML'''
                pass
            def handle_decl(self, data):
                '''Metodo invocato per le direttive HTML'''
```

```
                pass
```

```
In [6]: def fparse(fhtml):
            '''Esegue il parsing HTML del file fhtml e ritorna la radice dell'albero .'''
            with open(fhtml) as f: html=f.read()
            parser = _MyHTMLParser()
            parser.feed(html)
            return parser.root
```

```
In [7]: doc = fparse('semplice.html')
```

```
In [8]: print(doc.to_string())
```

```
<html>
<body>
<h1>Un Semplice Documento</h1>
<p>Un paragrafo con testo
<em>enfatizzato</em> e <strong>molto enfatizzato</strong>.
</p>
<p>Segue il logo dell'
<a href="http://en.wikipedia.org/wiki/HTML5">HTML 5</a>.</p>
<img src="img_logo.png">
</body>
</html>
```

```
In [9]: doc.print_tree()
```

```
<html>
  _text_ '\n'
  <body>
    _text_ '\n'
    <h1>
      _text_ 'Un Semplice Documento'
    _text_ '\n'
    <p>
      _text_ 'Un paragrafo con testo\n'
      <em>
        _text_ 'enfatizzato'
      _text_ ' e '
      <strong>
        _text_ 'molto enfatizzato'
      _text_ '.\n'
    _text_ '\n'
    <p>
      _text_ "Segue il logo dell'\n"
      <a>
        _text_ 'HTML 5'
      _text_ '.'
```

```
    _text_ '\n'
    <img>
    _text_ '\n'
  _text_ '\n'
```

Ora che abbiamo l'albero di parsing possiamo eseguire operazioni sul documento.

Possiamo implementare queste operazioni sia come metodi che come funzioni. Nel seguito per semplicità useremo le funzioni.

In [10]:
```
# contiamo quanti nodi di sono nel sottoalbero radicato in nodo

def count(nodo):
    '''Ritorna il numero di nodi dell'albero di questo nodo'''
    tot = 1
    if nodo.tag!='_text_':
        for figlio in nodo.content:
            tot += count(figlio)
    return tot

count(doc)
```

Out[10]: 25

In [11]:
```
#calcoliamo l'altezza dell'albero con radice nodo

def height(nodo):
    '''Ritorna l'altezza dell'albero con radice questo nodo, cioè il massimo numero di
    in un cammino radice-foglia'''
    h = 1
    if nodo.tag!='_text_':
        for figlio in nodo.content:
            h = max(h, height(figlio) + 1)
    return h

height(doc)
```

Out[11]: 5

In [12]:
```
#cerchiamo nel sottoalbero i nodi che hanno quel tag

def find_by_tag(nodo, tag):
    '''Ritorna una lista dei nodi che hanno il tag'''
    ret = []
    if nodo.tag == tag: ret += [nodo]
    if nodo.tag!='_text_':
        for figlio in nodo.content:
            ret += find_by_tag(figlio,tag)
    return ret
```

```
         find_by_tag(doc,'a')
```

```
Out[12]: [<__main__.HTMLNode at 0x10a219320>]
```

```
In [13]: # stampiamo i sottoalberi radicati nei nodi con quel tag

         for nodo in find_by_tag(doc,'a'):
             print(nodo.to_string())
```

```
<a href="http://en.wikipedia.org/wiki/HTML5">HTML 5</a>
```

```
In [22]: # stampiamo i sottoalberi radicati nei nodi con quel tag

         for nodo in find_by_tag(doc,'p'):
             print(nodo.to_string(),'\n','\n')
```

```
<p>Un paragrafo con testo
<em>enfatizzato</em> e <strong>molto enfatizzato</strong>.
</p>


<p>Segue il logo dell'
HTML 5.</p>
```

1.2 Possiamo anche modificare l'albero cancellando ad esempio tag di un certo tipo e facendo in modo che i tag figli dei tag cancellati divengano figli del padre di questi

```
In [23]: def remove_by_tag(nodo, tag):
             '''Rimuove dall'albero tutti i nodi con il tag, esclusa la radice,
             cioè il nodo su cui è invocato il metodo.'''
             if nodo.tag=='_text_': return
             for figlio in nodo.content:
                 remove_by_tag(figlio,tag)
             newcontent = []
             for figlio in nodo.content:
                 if figlio.tag == tag:
                     if nodo.tag!='_text_':
                         newcontent += figlio.content
                 else:
                     newcontent += [figlio]
             nodo.content = newcontent
```

```
In [24]: remove_by_tag(doc,'a')
         print(doc.to_string())
```

```
<html>
<body>
<h1>Un Semplice Documento</h1>
<p>Un paragrafo con testo
<em>enfatizzato</em> e <strong>molto enfatizzato</strong>.
</p>
<p>Segue il logo dell'
HTML 5.</p>
<img src="img_logo.png">
</body>
</html>
```

```
In [25]: #definiamo una funzione che rimuove dal primo file tutti i tag
         # del tipo specificato e salva il testo html ottenuto nel secondo file

         def rimuovi(fname,tag,fname1):
             '''rimuove dal file fname tutti i tag e lo registra in fname1'''
             doc= fparse(fname)
             remove_by_tag(doc,tag)
             with open(fname1,'w') as f: f.write(doc.to_string())

In [26]: rimuovi('semplice.html','a','semplice1.html')

In [27]: rimuovi('semplice.html','em','semplice1.html')
```

1.3 Grazie all'albero, possiamo fare delle statistiche sul documento di testo:

```
In [28]: def statistiche(fname):
             '''Stampa alcune statistiche della pagina html specificata'''
             tree= fparse(fname)
             print('Numero di nodi:', count(tree))
             print('Altezza:', height(tree))
             print('Numero di links:', len(find_by_tag(tree,'a')))
             print('Numero di immagini:', len(find_by_tag(tree,'img')))

         statistiche('semplice.html')
```

```
Numero di nodi: 25
Altezza: 5
Numero di links: 1
Numero di immagini: 1
```

1.4 Tutto questo ha più senso se il documento di testo è grande:

```
In [21]: statistiche('python_wiki.html')
```

```
Numero di nodi: 9285
Altezza: 16
Numero di links: 1396
Numero di immagini: 23
```