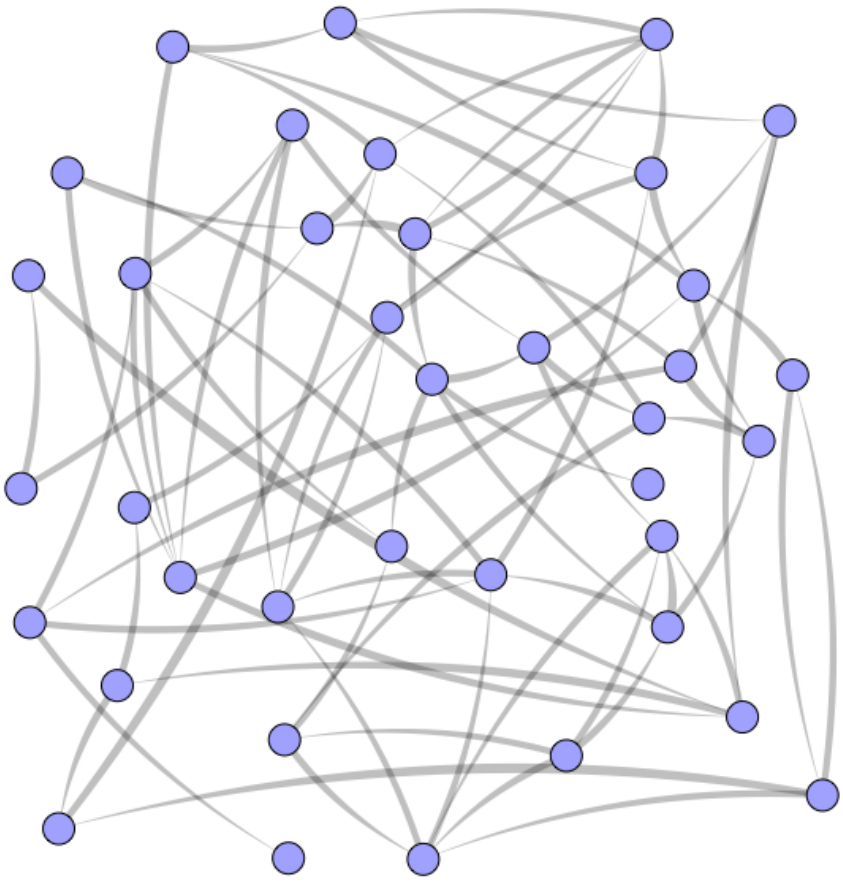
WEB CRAWLING

Il *Web Crawling* è la visita delle pagine del Web partendo da una o più pagine e seguendo i link in esse contenuti. Uno degli scopi più importanti è l'effettuazione di ricerche (ad es. Google, si veda WebCrawler. I programmi di questa ultima lezione sono nel pacchetto.

Il Web crawling è semplicemente la visita del grafo del Web i cui nodi sono le pagine e gli archi sono i *link* tra le pagine. Ovviamente, i link sono archi che hanno un verso, da una pagina verso un'altra.



Si inizia da un URL, si scarica la pagina relativa all'URL e si estraggono tutti i link contenuti in essa, poi si cerca di scaricare le pagine relative a tutti i link e per ognuna si ripete la stessa procedura. Per evitare di scaricare più volte la stessa pagina, si mantiene l'insieme dei link che sono già stati archiviati.

Un programma che effettua il Web crawling si chiama *Web Crawler*. Per implementarlo, anche se molto semplificato, dobbiamo saper scaricare il contenuto di una pagina (tipicamente in HTML), dobbiamo saper estrarre i link da una pagina (tramite un opportuno parsing) e infine dobbiamo saper usare questi strumenti per implementare una visita del grafo del Web.

SCARICARE UNA PAGINA

Scaricare una pagina Web dato il suo URL, pur essendo un'operazione concettualmente molto semplice, dipende da parecchi dettagli che ne rendono piuttosto delicata una realizzazione soddisfacente. Per scaricare una pagina bisogna effettuare una richiesta a un server (il cui indirizzo è contenuto nell'URL). La richiesta è effettuata tramite il protocollo HTTP (*HyperText Transfer Protocol*, si veda ad es. HTTP e comprende parecchie informazioni oltre all'URL, come ad esempio, il formato accettato (ad es. `text/html`), le codifiche del contenuto (ad es. `gzip`), le codifiche dei caratteri (chiamate `charset`, ad es. `UTF-8`), e molte altre. Se queste informazioni non vengono fornite nella richiesta, il server può assumere che il *client*, cioè, chi effettua la richiesta, accetta qualsiasi formato, codifica, ecc. La funzione `urlopen()` del modulo `urllib2` che abbiamo già usato nella sua forma più semplice, permette di specificare anche queste informazioni aggiuntive. Basta creare un opportuno oggetto `Request` che "incolla" ad un URL queste informazioni sotto forma di un dizionario che rappresenta l'*header* della richiesta.

Ecco quindi una implementazione preliminare, chiamata appunto `loadpage1()`, che si accontenta di ottenere l'oggetto di tipo `file` che ci restituisce `urlopen()` e di stampare le informazioni collegate alla pagina che il server ritorna (l'attributo `headers`). Anche quest'ultime sono rappresentate tramite un dizionario e possiamo vederne degli esempi provando la funzione su alcuni URL.

```
import urllib2 as ul

# Header per le richieste HTTP
HEADER = {'User-Agent':"Mozilla/5.0",
          'Accept':"text/html;q=1.0,*;q=0",
          'Accept-Encoding':"identity;q=1.0,*;q=0"}

def loadpage1(url):
    req = ul.Request(url, headers=HEADER)  # Crea la richiesta per l'url
    f = ul.urlopen(req, timeout=5)         # Esegue la richiesta
    print url, f.headers             # Stampa gli headers della risposta
```

Se la proviamo su alcuni URL,

```
load_page_1('http://www.python.org')
load_page_1('http://www.agi.it/borsa')
load_page_1('http://www.nonesiste.com')
```

Otteniamo:

```
http://www.python.org Date: Fri, 03 Jan 2014 15:35:01 GMT
Server: Apache/2.2.16 (Debian)
Last-Modified: Tue, 31 Dec 2013 15:02:37 GMT
ETag: "105800d-4f5d-4eed5d9d67d40"
"Accept-Ranges: bytes
Content-Length: 20317
Vary: Accept-Encoding
Connection: close
Content-Type: text/html

http://www.agi.it/borsa Server: nginx
Date: Fri, 03 Jan 2014 15:35:02 GMT
Content-Type: text/html
Transfer-Encoding: chunked
Connection: close
Vary: Accept-Encoding
Content-Encoding: gzip
Expires: Fri, 03 Jan 2014 15:39:02 GMT
Cache-Control: max-age=240

Traceback (most recent call last):
   . . .
    raise HTTPError(req.get_full_url(), code, msg, hdrs, fp)
HTTPError: HTTP Error 999: AW Special Error
```

Notiamo che ci sono chiavi come `Content-Type` e `Content-Encoding` che però non compaiono sempre: `Content-Encoding` compare solo se la pagina ritornata è in forma codificata, tipicamente compressa. Il terzo URL ha prodotto un errore in quanto è inesistente. Infatti, quando `urlopen()` incontra un errore, e ci possono essere varie cause, solleva una specifica eccezione (*Exception*). Ma noi non vogliamo che la nostra funzione che scarica una pagina possa mandare in crash il programma. Per evitare ciò, Python ha un costrutto che permette di catturare le eccezioni. È il `try` la cui forma più semplice è la seguente:

```
try:
    <istruzioni che possono produrre eccezioni>
except:
    <istruzioni eseguite solo se si verifica un'eccezione>
```

Oppure, se vogliamo sapere quale eccezione si è verificata:

```
try:
    <istruzioni che possono produrre eccezioni>
except Exception as ex:  # ex e' l'oggetto relativo
                         # all'eccezione
    <istruzioni eseguite solo se si verifica un'eccezione>
```

In questo modo nel blocco di `except`, l'oggetto `ex` è l'eccezione verificatesi. Il tipo `Exception` comprende

tutti i tipi di eccezione.

Nella seconda versione `loadpage2` usiamo il costrutto `try` e inoltre stampiamo solamente le informazioni che ci interessano che sono `Content-Type` e `Content-Encoding`:

```
def loadpage2(url):
    print url
    try:
        req = ul.Request(url, headers=HEADER)
        f = ul.urlopen(req, timeout=5)
        hh = f.headers
        for k in ('Content-Type','Content-Encoding'):
            print k+':', (hh[k] if k in hh else None)
    except Exception as ex:
        print ex
```

Provandola su alcuni URL,

```
loadpage2('http://www.python.org')
loadpage2('http://www.agi.it/borsa')
loadpage2('http://www.nonesiste.com')
loadpage2('http://www.google.com')
```

Otteniamo:

```
http://www.python.org
Content-Type: text/html
Content-Encoding: None
http://www.agi.it/borsa
Content-Type: text/html
Content-Encoding: gzip
http://www.nonesiste.com
HTTP Error 999: AW Special Error
http://www.google.com
Content-Type: text/html; charset=UTF-8
Content-Encoding: None
```

Ora l'eccezione sollevata dal terzo URL è stata catturata e non manda in crash il programma.

Dobbiamo modificare la nostra funzione, perché deve ritornare la pagina scaricata e non deve stampare nulla. Inoltre, se compare la chiave `Content-Encoding` non vogliamo ritornare la pagina perchè il nostro Web Crawler semplificato non tratta pagine compresse. Si noti che nel header della nostra richiesta c'è `'Accept-Encoding':"identity;q=1.0,*;q=0` che significa che il nostro `client` non accetta alcun tipo di codifica. Nonostante questo, alcuni server inviano ugualmente pagine compresse, come nel caso dell'URL `http://www.agi.it/borsa`. Siccome la nostra funzione può fallire nello scaricare la pagina, a causa di errori di tipo HTTP o di codifica, conveniamo che ritorni una tripla `(url, page, ex)` che in `url` contiene l'URL

richiesto, in `page`, se la pagina è stata scaricata senza errori, contiene la pagina in unicode (questo per rendere più agevole il successivo parsing), altrimenti sarà `None`, e infine `ex` contiene la descrizione sotto forma di stringa dell'eventuale eccezione verificatesi o `None` altrimenti.

```
def loadpage3(url):
    try:
        req = ul.Request(url, headers=HEADER)
        f = ul.urlopen(req, timeout=5)
        hh = f.headers
        ce = 'Content-Encoding'
        if ce in hh:    # La pagina e' compressa, non c'interessa
            return (url, None, ce+': '+hh[ce])
        # Assumiamo che l'encoding della pagina sia UTF-8
        page = unicode(f.read(), encoding='utf-8')
        return (url, page, None)
    except Exception as ex:
        return (url, None, str(ex))
```

Si osservi che per la codifica in unicode abbiamo assunto che la pagina sia stata ritornata nella codifica `utf-8`, come vedremo presto, è un'assunzione azzardata. Scriviamo una piccola funzione di test `test(lp, *urls)` che ci pemette di provare diverse funzioni di caricamenti pagina `lp` su uno o più URL `urls`:

```
import urlparse as up

def test(lp, *urls):
    for url in urls:
        if not up.urlsplit(url).scheme:  # Se non ha lo schema,
            url = 'http://' + url        # lo aggiungiamo
        url, page, ex = lp(url)
        if page != None:
            print url, 'LOADED', len(page)
        else:
            print url, ex
```

La funzione `urlsplit()` del modulo `urlparse` suddividde un URL nelle sue componenti `scheme://netloc/path;parameters?query#fragment` e ognuna è ritornata in un attributo con lo stesso nome (se non è presente il valore è la stringa vuota). Qui abbiamo usato `urlsplit()` per determinare se lo schema è presente, se non lo è, aggiungiamo quello di default che è `http`. Ed ecco un test:

```
test(loadpage3, 'www.google.com','www.agi.it/borsa','www.nonesiste.com')
```

che produce il risultato:

```
http://www.google.com LOADED 44677
http://www.agi.it/borsa Content-Encoding: gzip
http://www.nonesiste.com HTTP Error 999: AW Special Error
```

Ma provando `test(loadpage3, 'www.twiki.di.uniroma1.it')` otteniamo:

```
 http://www.twiki.di.uniroma1.it 'utf8' codec can't decode byte 0xe0 in position 9267:
invalid continuation byte
```

Questo significa che l'assunzione che la pagina scaricata avesse la codifica `utf-8` era errata. Per rimediare dobbiamo prima di tutto tener in conto che non tutti i server dichiarano il `charset` usato e anche se aggiungessimo alla nostra richiesta che accettiamo solamente pagine in `utf-8`, molti server non rispetterebbero la richiesta. Allora, quello che possiamo fare è leggere il `charset` nel caso il server lo dichiari e se non lo dichiara provare le due codifiche più comuni che sono `utf-8` e `latin-1`. Siamo così pronti per scrivere la versione finale della nostra funzione:

```
def loadpage(url):
    try:
        req = ul.Request(url, headers=HEADER)
        f = ul.urlopen(req, timeout=5)
        hh = f.headers
        ce = 'Content-Encoding'
        if ce in hh:
            return (url, None, ce+': '+hh[ce])
        charsets = ['utf-8', 'latin-1']
        if 'Content-Type' in hh:
            ct = hh['Content-Type'].lower()
            i = ct.find('charset=')
            if i >= 0:
                charsets.insert(0, ct[i+len('charset='):])
        page = f.read()
        for enc in charsets:
            try:
                page = unicode(page, encoding=enc)
                break
            except: continue
        else: return (url, None, 'Encoding Error')
        return (url, page, None)
    except Exception as ex:
        return (url, None, str(ex))
```

Il `for` relativo ai `charsets` prova i vari tipi di codifica fino a che non ne trova uno che va bene, nel qual caso esce dal `for`. Se non ne trova nessuno rinuncia a decodificare la pagina e ritorna `Encoding Error`. Facciamo un test relativo all'URL precedente:

```
test(loadpage, 'www.twiki.di.uniroma1.it')
```

E otteniamo:

```
http://www.twiki.di.uniroma1.it LOADED 69135
```

PARSING DEI LINKS

Ora che sappiamo scaricare le pagine, possiamo passare all'estrazione dei links. Useremo il modulo `html` che già conosciamo. Per prima cosa implementiamo una classe per il parsing con un metodo `getlinks()` che ritorna la lista dei links. Consideriamo solamente i links contenuti nei tag `a`:

```
class HTMLNode(object):
    def __init__(self, tag, attr, content, closed=True):
        self.tag = tag
        self.attr = attr
        self.content = content
        self.closed = closed

    def getlinks(self, d=0):
        links = []
        if (self.tag == 'a' and 'href' in self.attr and
            self.attr['href'] != None):
            links.append(self.attr['href'])
        if self.tag != '_text_' and d < 100:
            for c in self.content:
                links.extend(c.getlinks(d + 1))
        return links
```

Abbiamo dovuto mettere un limite sulla profondità della ricorsione perché alcune pagine malformate producono alberi di parsing con profondità molto elevate o con dei cicli. Vediamo ora la funzione che data una pagina ritorna un insieme di link accettabili. La funzione `getlinks()` permette di ritornare solo i link che appartengono allo stesso dominio dell'URL della pagina e/o di filtrare quelli che hanno la componente query.

```
import html, os
import urlparse as up
from loadpage import loadpage


def getlinks(url, page, domain=False, noquery=False):
    '''Ritorna i links della pagina page. Se domain e' True, esclude i
    links con dominio diverso da quello dell'URL url. Se noquery e' True,
    esclude i links con la componente query.'''
    try:
        parsed = html.parse(page, HTMLNode)
    except Exception as ex:
        return (None, str(ex))
```

```
    if parsed == None:
        return (None, 'HTML Parsing Error')
    linkset = set()
    for link in parsed.getlinks():
        try:
            plink = up.urlsplit(link)
        except:                     # URL malformato
            continue
        ext = os.path.splitext(plink.path)[1].lower()
        if (plink.scheme.lower() in ('http','https','ftp','')
            and (plink.netloc or plink.path)
            and ext in ('','.htm','.html')):
            if noquery and plink.query: continue
            if not plink.netloc:
                link = up.urljoin(url, link)
                plink = up.urlsplit(link)
            else:
                if domain:
                    if plink.netloc != up.urlsplit(url).netloc:
                        continue
                if not plink.scheme:
                    link = 'http://' + link
                    plink = up.urlsplit(link)
            linkset.add(plink.geturl())
    return (linkset, None)
```

Implementiamo una funzione per effetture qualche test:

```
def test(*urls):
    for url in urls:
        if not up.urlsplit(url).scheme: url = 'http://' + url
        url, page, ex = loadpage(url)
        if page != None:
            print url, 'LOADED', len(page)
            links, ex = getlinks(url, page)
            if links != None:
                print '  Numero Links:', len(links)
            else:
                print ex
        else:
            print url, ex

test('www.python.org')
```

Ed ecco il risultato:

```
http://www.python.org LOADED 20313
  Numero Links: 69
```

WEB CRAWLER

Siamo pronti per mettere tutto insieme e implementare un semplice Web Crawler. Per far sì che sia versatile e possa essere lanciato e fermato a piacimento, lo implementiamo con una classe. Così può facilmente mantenere lo stato del crawling che è l'URL di partenza (l'attributo `url`), l'archivio di tutti gli URL estratti (l'attributo `all_links`) e la lista di quelli che devono ancora essere scaricati (l'attributo `toload`). In questa versione abbiamo preferito scegliere il prossimo URL da scaricare in modo random tra tutti quelli in `toload`. Questo per dare una maggiore varietà alle pagine scaricate. Siccome i link in `toload` sono mantenuti in una lista nell'ordine compatibile con la visita in ampiezza, se il prossimo link è sempre il primo della lista il crawler esegue proprio una visita in ampiezza.

```
import urlparse as up
import random
from loadpage import loadpage
from getlinks import getlinks

class Crawler(object):
    def __init__(self):
        '''Inizializza il web crawler'''
        self.domain, self.noquery = False, False

    def setUrl(self, url):
        '''Imposta l'url di partenza del crawling. Per eseguire il
        crawling invocare ripetutamente il metodo getPage()'''
        url = url.strip()
        if not up.urlsplit(url).scheme:   # Se manca lo schema,
            url = 'http://' + url         # aggiungilo
        self.url = url
        self.all_links = set([url])  # Archivio di tutti i links
        self.toload = [url]          # Lista dei links da scaricare

    def getPage(self):
        '''Effettua un passo del crawling impostato tramite il metodo
        setUrl(). Se il crawling e' terminato, ritorna None. Se non e'
        terminato, ritorna una tripla (url, page, err) dove url e' l'URL
        della pagina che ha tentato di scaricare (puo' essere None nel
        caso non sia ancora pronta una pagina), se questa e' stata
        scaricata, page e' il contenuto della pagina, altrimenti e' None
        e err e' una stringa che descrive l'errore che si e' verificato.
        Anche se la pagina e' stata scaricata err e' non None quando si
        verifica un errore durante il parsing dei links.'''
        toload = self.toload
        if len(toload) == 0:        # Se non ci sono link da scaricare,
            self.url = None         # il crawling e' terminato
            return None
```

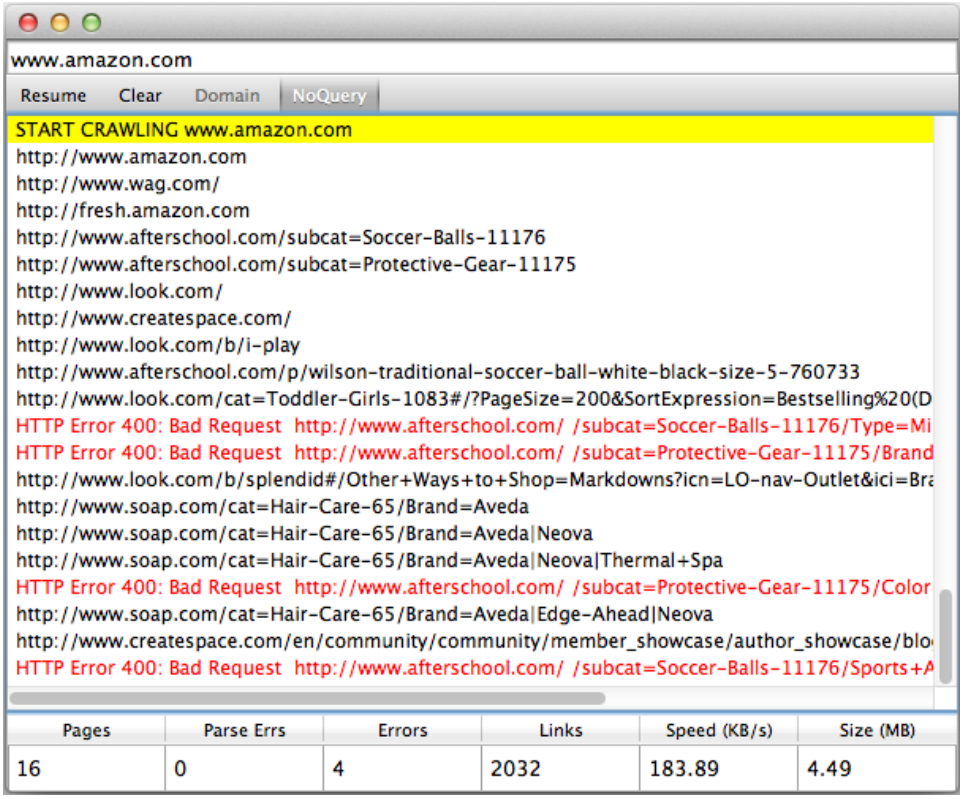
```
            i = random.randrange(0, len(toload))  # Sceglie un link random
            url = toload[i]                       # tra quelli da scaricare
            del toload[i]
            url, page, err = loadpage(url)     # Cerca di scaricare il link
            if page:    # Se e' stato scaricato, cerca i link della pagina
                links, perr = getlinks(url, page, self.domain, self.noquery)
                if links != None:              # Se il parsing ha successo,
                    # Rimuovi dai link della pagina quelli gia' archiviati
                    links.difference_update(self.all_links)  # e aggiungi
                    self.all_links.update(links)  # i rimanenti all'archivio
                    self.toload.extend(links)     # e a quelli da scaricare
                else:
                    err = perr
            return (url, page, err)

        def config(self, domain=None, noquery=None):
            '''Se domain e' True, esclude i links con dominio diverso da quello
            dell'URL url. Se noquery e' True, esclude i links con la componente
            query.'''
            if domain != None: self.domain = domain
            if noquery != None: self.noquery = noquery

        def nLinks(self):
            '''Ritorna il numero totale di links nell'archivio, sia gia'
            scaricati che ancora da scaricare'''
            return len(self.all_links)
```

È facile scrivere una semplice funzione che usa la classe `Crawler` per fare un crawling a partire da un dato URL:

```
def test(start_url, maxpages=10):
    crawler = Crawler()
    crawler.setUrl(start_url)
    print 'START CRAWLING:', start_url
    while maxpages > 0:
        p = crawler.getPage()
        maxpages -= 1
        if p != None:
            url, page, err = p
            if url:
                if page != None:
                    print 'LOADED',
                    if err != None: print err,
                    print len(page), url
                else:
                    print err, url
    print 'END CRAWLING:', start_url
```

Il modulo `vcrawling` implementa una funzione `view` che preso in input un oggetto compatibile con i metodi della classe `Crawler`, fornisce una GUI che gestisce il crawling:

```
from vcrawler import view
view(Crawler())
```



WEB CRAWLER CONCORRENTE

Siccome le operazioni che accedono alla rete inoltrando richieste a server remoti perdono molto tempo aspettando le risposte dei server, la precedente implementazione della classe `Crawler` è inefficiente. Qualsiasi Web browser sfrutta il tempo d'attesa inoltrando altre richieste per altre pagine. Per fare qualcosa di simile dobbiamo usare necessariamente più *processi* (o *thread*) che sono eseguiti in parallelo o in modo concorrente. Ogni processo si occupa di scaricare una pagina differente. In generale, la corretta implementazione di programmi concorrenti è piuttosto complessa e delicata. Nel nostro caso però i compiti che i processi devono eseguire sono semplici e anche le loro dipendenze lo sono. La libreria di Python ha parecchi moduli che permettono di creare e gestire processi concorrenti. Però questi non sono perfettamente compatibili con Qt/PySide, quindi useremo un nostro modulo `qmp` che internamente usa le classi di Qt/PySide per implementare una classe `Pool` simile a quella del modulo `multiprocessing` della librearia standard di Python. La classe `Pool` crea e gestisce un gruppo o pool di *n* processi, dove *n* può

essere deciso a piacimento. Una volta creato un pool si può pianificare un compito da esegure tramite il metodo `apply_async(func, args)` che esegue la funzione `func`, con input dato da `args` (che deve essere una tupla), in un thread separato. Il metodo ritorna un oggetto che ha un metodo `ready()`, che ritorna `True` quando la risposta del relativo compito è pronta, e un metodo `get()` che ritorna la risposta (cioè l'output della funzione `func` su input `args`). Se si invoca `get()` prima che la risposta sia pronta questa blocca fino a che la risposta è pronta.

```
import qmp, random
import urlparse as up
from loadpage import loadpage
from getlinks import getlinks

class Crawler(object):
    PROCESSES = 10       # Massimo numero di processi concorrenti

    def __init__(self):
        '''Inizializza il web crawler'''
        self.domain, self.noquery = False, False

    def setUrl(self, url):
        '''Imposta l'url di partenza del crawling. Per eseguire il
        crawling invocare ripetutamente il metodo getPage()'''
        self.pool = qmp.Pool(Crawler.PROCESSES)  # Pool dei processi
        self.tasks = []                      # Lista dei tasks attivi
        url = url.strip()
        if not up.urlsplit(url).scheme:
            url = 'http://' + url
        self.url = url
        self.all_links = set([url])
        self.toload = [url]

    def getPage(self):
        '''Effettua un passo del crawling impostato tramite il metodo
        setUrl(). Se il crawling e' terminato, ritorna None. Se non e'
        terminato, ritorna una tripla (url, page, err) dove url e' l'URL
        della pagina che ha tentato di scaricare (puo' essere None nel
        caso non sia ancora pronta una pagina), se questa e' stata
        scaricata, page e' il contenuto della pagina, altrimenti e' None
        e err e' una stringa che descrive l'errore che si e' verificato.
        Anche se la pagina e' stata scaricata err e' non None quando si
        verifica un errore durante il parsing dei links.'''
        toload, tasks = self.toload, self.tasks
        if len(toload) == 0 and len(tasks) == 0:  # Se non ci sono ne'
            self.url = None                       # link da scaricare ne'
            return None                           # task attivi, termina
        url, page, err = None, None, None
        for i in range(len(tasks)):       # Controlla i task attivi per
            if tasks[i].ready():          # vedere se uno e' completato
                url, page, err = tasks[i].get()  # Ottieni il risultato e
```

```
                del tasks[i]                    # rimuovilo dalla lista
                if page:
                    links, perr = getlinks(url, page, self.domain,
                                           self.noquery)
                    if links != None:
                        links.difference_update(self.all_links)
                        self.all_links.update(links)
                        self.toload.extend(links)
                    else:
                        err = perr
                break
        # Se c'e' almeno un processo disponibile e ci sono dei link da
        if len(tasks) < Crawler.PROCESSES and len(toload): # scaricare,
            i = random.randrange(0, len(toload))
            newurl = toload[i]
            del toload[i]
            # Crea un task asincrono per scaricare un nuovo link
            tasks.append(self.pool.apply_async(loadpage, (newurl,)))
        return (url, page, err)

    def config(self, domain=None, noquery=None):
        '''Se domain e' True, esclude i links con dominio diverso da quello
        dell'URL url. Se noquery e' True, esclude i links con la componente
        query.'''
        if domain != None: self.domain = domain
        if noquery != None: self.noquery = noquery

    def nLinks(self):
        '''Ritorna il numero totale di links nell'archivio, sia gia'
        scaricati che ancora da scaricare'''
        return len(self.all_links)
```

Se si prova questo crawler si noterà che la velocità è significativamente superiore.