Programmazione di rete con Python: le librerie Django e Twisted

Vittorio Palmisano <vpalmisano@gmail.com>

08/10/2013

Sommario

Sommario

Introduzione a Python

- linguaggio interpretato
- tipizzazione dinamica
- orientato ad oggetti e modulare
- shell interattiva
- portabile su diverse piattaforme (Linux, Win32, Symbian, ...)
- libero, con un gran numero di moduli già disponibili

L'interprete

modalità di utilizzo:

Interattivo

Consente di scrivere istruzioni ed eseguirle all'interno di una shell:

```
$ python
>>>
```

Non interattivo

Esegue le istruzioni presenti in un file:

```
$ python nomefile.py
```

IPython: un interprete avanzato

- autocompletamento
- salvataggio sessioni
- esecuzione di comandi da shell sh

Tipi di dato numerici

- Differenze tra tipizzazione forte, debole e **dinamica**
- I tipi di dati di base:

int

```
>>> a = 10
>>> print a / 3
3
```

float

```
>>> a = 10
>>> print a / 3.0
3.3333333333333335
```

long

```
>>> a = 1234235235235
>>> print a
1234235235235L
```

complex

```
>>> a = 10 + 2j
>>> print a.imag, a.real
2.0 10.0
```

Sommario

Le funzioni

- L'**indentazione** deve essere consistente
- Le variabili hanno visibilità all'interno del **blocco** della funzione
- Il passaggio dei parametri avviene sempre per **riferimento** (infatti ogni variabile è un oggetto)
- Alcune funzioni predefinite: **ord()**, **chr()**, **hex()**, **dir()**, **type()**

Creazione di nuove funzioni con **def**

```
>>> def somma(arg1, arg2):
>>>     "Effettua la somma di due numeri"
>>>     return arg1 + arg2
>>>
>>>     print somma(3, 4)
7
```

Strutture di controllo e di flusso

Condizionali: **if ... elif ... else**

```
>>> if colore == 'rosso':
...       print 'Hai scelto rosso'
...   elif colore == 'blu':
...       print 'Hai scelto blu'
...   else:
...       print 'Colore sconosciuto'
```

Iterative: **for**, **while**

```
>>> for elemento in [1, 2, 3.2, 4,1, 'ciao']:
>>>     print "Valore corrente:", elemento
>>>
>>> while variabile > 0:
>>>     print variabile
```

Sommario

Fondamenti di programmazione a oggetti (1/2)

- Classi e oggetti
- Metodi e proprietà
- In Python tutto è un **oggetto**
- Creazione di nuove classi con **class**
  - il metodo di default **__init__()**
  - metodi speciali: **__str__**, **__doc__**
  - variabili pubbliche e private: gestire l'accesso con i metodi **__getattr__()** e **__setattr__()**
- Ereditarietà semplice e multipla

Fondamenti di programmazione a oggetti (2/2)

Esempio di creazione di una classe

```
class Rettangolo:
    "Classe che rappresenta un rettangolo"
    def __init__(self, larghezza, altezza):
        self.larghezza, self.altezza = larghezza, altezza
    def __str__(self):
        return 'Rettangolo'
    def area(self):
        return self.larghezza * self.altezza
```

Esempio di ereditarietà semplice

```
class Quadrato(Rettangolo):
    "Classe che rappresenta un quadrato"
    def __init__(self, lato):
        self.larghezza, self.altezza = lato, lato
    def __str__(self):
        return 'Quadrato'
```

Sommario

Le stringhe

Creazione, concatenazione, formattazione

```
>>> stringa = 'ciao'
>>> print stringa + " a tutti"
ciao a tutti
>>> print len(stringa)
4
>>> print '%s:  %d %f' %('ciao', 12, 0.32)
ciao:  12 0.320000
```

Alcuni metodi

```
>>> s = "ciao a tutti"
>>> s.split()
['ciao', 'a', 'tutti']
>>> s.replace('ciao', 'buongiorno')
'buongiorno a tutti'
>>> " \t\n inizio \t fine \n\n".strip()
'inizio \t fine'
```

Le liste

Il tipo list

```
>>> l = [1, 3, 2.5, 'ciao']
>>> print l[2]
2.5
>>> l = l + [3, 4.1]
>>> print l
[1, 3, 2.5, 'ciao', 3, 4.1]
>>> print range(0, 5)
[0, 1, 2, 3, 4]
>>> print l[0:3]
[1, 3, 2.5]
>>> print [2:4]
[2.5, 'ciao', 3, 4.1]
>>> print [4:]
[3, 4.1]
```

Alcuni metodi

```
>>> l = [3, 2.5, 'ciao', 1]
>>> l.append(12)
>>> l.remove('ciao')
>>> print l
[3, 2.5, 1, 12]
>>> l.sort()
>>> print l
[1, 3, 2.5, 12]
```

I dizionari

Creazione e uso

```
>>> d = {'giovanni': 13.7, 'marco': 24, 'giuseppe': 'roma',
'antonio': 'bari'}
>>> d['marco']
24
>>> d['francesco'] = 18.9
```

Alcuni metodi

```
>>> print d.keys()
['francesco', 'giovanni', 'giuseppe', 'marco', 'antonio']
>>> print d.values()
[18.899999999999999, 13.699999999999999, 'roma', 24, 'bari']
>>> print d.items()
[('francesco', 18.899999999999999), ('giovanni',
13.699999999999999), ('giuseppe', 'roma'), ('marco', 24),
('antonio', 'bari')]
```

Sommario

Moduli e package

Ogni file **.py** contenente istruzioni è un **modulo**:

Importare un modulo

```
>>> import nome_file
>>> from nome_file import funzione, Classe
```

Ogni directory contenente file **.py** e un file **__init__.py** è un **package**:

Creazione di package:

```
package/__init__.py
package/nome_modulo.py
package/altro_modulo.py
>>> import package.nome_file
>>> from package.nome_file import *
```

Alcuni moduli standard

- Accesso ai file: la classe **file**
- Le librerie di sistema **os** e **sys**
- Operazioni avanzate sulle stringhe (*espressioni regolari*): il modulo **re**
- Gestione di dati temporali: i moduli **time** e **datetime**
- Salvare gli oggetti su file: il modulo **pickle**
- Accesso a Internet: i moduli **urllib2** e **smtplib**
- Le librerie per il calcolo scientifico e il plotting di dati: **scipy**, **numpy** e **matplotlib**

Gestire le condizioni di errore: le eccezioni

Catturare un'eccezione: **try ... except**

```
>>> try:
>>>     print 1 / 0
>>> except ZeroDivisionError:
>>>     print 'Impossibile dividere per 0'
```

Causare un'eccezione con **raise**

```
>>> raise Exception('BUG!')
Traceback (most recent call last):
  File "<stdin>", line 1, in ?
Exception:  BUG!
```

Sommario

Django: caratteristiche principali

- Sviluppato per la gestione di siti per la gestione di news (World Online)
  - Facilità di utilizzo
  - Velocità di sviluppo
  - Personalizzazione elevata
- DRY (*Don't Repeat Yourself*) principle
- Approccio Model-View-Controller (*MVC*)

View	Interfaccia utente (client side): **HTML**, **CSS**, **JS**
Controller	Controllore (server side): **Python**
Model	Applicazione (database): **SQLite**, **MySQL**, **PGSQL**,...

Sommario

Creazione di un nuovo progetto (1/2)

- Creare un nuovo progetto:
  :~$ django-admin **startproject** mysite

```
mysite/
    __init__.py
    settings.py
    urls.py
manage.py
```

- Lanciare il server di sviluppo:
  :~$ **python manage.py** runserver

Creazione di un nuovo progetto (2/2)

- Modificare le impostazioni globali in **mysite/settings.py**
- `DATABASE ENGINE (postgresql, mysql, sqlite3, oracle)`
  - `DATABASE NAME`
  - `MEDIA_ROOT (percorso assoluto alla dir /media)`
  - `MEDIA_URL = '/media/'`
  - `MIDDLEWARE_CLASSES += ['django.middleware.locale.LocaleMiddleware']`
  - `INSTALLED_APPS += ['django.contrib.admin']`
- Aggiornare le ultime modifiche sul database:
  :~$ python manage.py **syncdb**

Creare e gestire una nuova applicazione

- Creazione:
  :~$ python manage.py **startapp** blog
  mysite/blog/
    __init__.py
    admin.py
    models.py
    views.py
- Aggiungere l'applicazione in **settings.INSTALLED_APPS**
- Creare la directory "**mysite/blog/templates**"
- Modifica di **mysite/blog/models.py**...
- Aggiornare le ultime modifiche sul database:
  :~$ python manage.py **syncdb**
- Altri comandi utilizzati per creare/aggiornare le table per ogni **app**:
  - **sqlcustom**: mostra le istruzioni SQL
  - **sqlclear**: mostra le istruzioni DROP TABLE
  - **sqlall**: mostra tutte le istruzioni SQL
  - **reset**: effettua tutte le istruzioni SQL

I Models

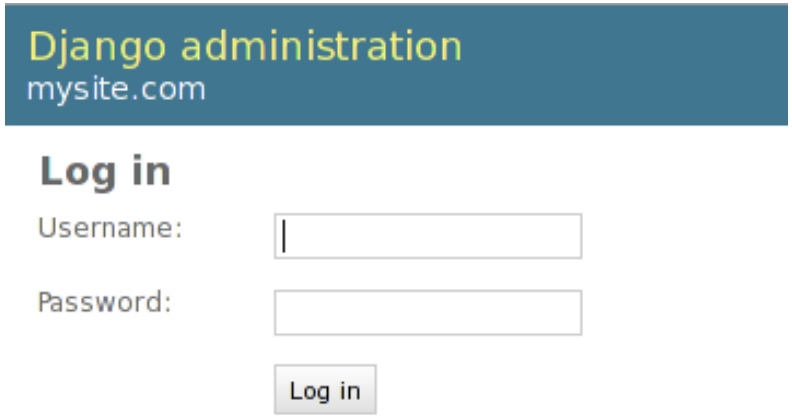
- Definiscono gli oggetti persistenti utilizzati dall'applicazione, memorizzati sul Database
- Esempio (**mysite/blog/models.py**):

```
from django.db import models

class Category(models.Model):
    name = models.CharField(max_length=100)

    def __unicode__(self):
        return self.name

class Article(models.Model):
    title = models.CharField(max_length=200)
    pub_date = models.DateTimeField('date published')
    content = models.TextField()
    category = models.ForeignKey(Category)

    def __unicode__(self):
        return self.title
```

L'interfaccia di amministrazione (1/2)

Installazione:

1. Aggiungere "**django.contrib.admin**" in **settings.INSTALLED_APPS**
2. :~$ python manage.py **syncdb**
3. Modificare **mysite/urls.py**
4. Modificare **mysite/blog/admin.py**



L'interfaccia di amministrazione (2/2)

Abilitare l'interfaccia di amministrazione per il nostro model:

```
class ArticleAdmin:
        fieldsets = (
            (None, {
              'fields':  ('title','content','category'),
             }),
            ('Date information', {
              'fields':  ('pub_date',),
              'classes':  'collapse',
             }),
        )
        list_display = ('title', 'pub_date', 'category')
        ordering = ('pub_date', )
        list_filter = ('pub_date', )
        search_fields = ('title', 'content')
        date_hierarchy = 'pub_date'
```

Gestione degli URL

mysite/urls.py:

```
from django.conf.urls.defaults import patterns, include, url

urlpatterns = patterns('',
    (r'^blog/$', 'mysite.blog.views.index'),
    (r'^blog/(?P<title>.+)/$', 'mysite.blog.views.article'),
)
```

Schema ottenuto:

URL	Funziona chiamata
/blog/	**index**(request)
/blog/titolo_articolo/	**article**(request, title)

Creazione delle View

mysite/blog/views.py:

```
from django.http import HttpResponse

def index(request):
    return HttpResponse("Blog index")

def article(request, title):
    return HttpResponse("Articolo: %s" %title)
```

I template: un primo esempio

templates/blog/index.html

```
{% if article_list %}
    <ul>
        {% for article in article_list %}
            <li>{{ article.title }}</li>
        {% endfor %}
    </ul>
{% else %}
    <p>Non ci sono articoli al momento.</p>
{% endif %}
```

templates/blog/article.html

```
{% if article %}
    <h1>{{ article.title }}</h1>
    <p>{{ article.content }}</p>
{% endif %}
```

Utilizzare i template nelle view

Esempio completo con query:

```
from django.shortcuts import render_to_response,
    get_object_or_404
from mysite.blog.models import Article

def index(request):
    article_list =
        Article.objects.all().order_by('-pub_date')[:5]
    return render_to_response('blog/index.html',
        {'article_list': article_list})

def article(request, title):
    a = get_object_or_404(Article, title=title)
    return render_to_response('blog/article.html',
        {'article': a})
```

Dettagli sulle view

- L'oggetto **request**:
  - **request.GET**, **request.POST**: dizionari degli argomenti passati via GET e POST
  - **request.FILES**: i file passati tramite upload HTTP
  - **request.COOKIES**: dizionario con i cookie
  - **request.session**: variabili di sessione per l'utente
- URL per applicazione:

```
from django.conf.urls.defaults import include
(r'^blog/', include('mysite.blog.urls'))
```

- URL semplificati:

```
urlpatterns = patterns('mysite.blog.views',
    (r'^blog/$', 'index'),
    (r'^blog/(?P<title>.+)/$', 'article'),
)
```

Dettagli sui model (1/2)

Argomenti aggiuntivi e alcuni tipi

Argomento	Significato
null	il campo può essere vuoto (db)
blank	il campo può essere vuoto (admin)
choices	una lista di valori (admin)
default	il valore predefinito
editable	modificabile (admin)
primary_key	il campo è chiave primaria
unique	il campo è unico (db+admin)
unique_for_date	unique_for_date="pub_date"

Tipo	Argomenti opzionali
DateField, DateTimeField	auto_now, auto_now_add
CharField	max_length
SlugField	prepopulate_from

Dettagli sui model (2/2)

Definire le **relazioni**:

- **Many-to-One**: es. Article(N) → Category(1)

```
class Article(models.Model):
    ...
    category = models.ForeignKey(Category)
```

- **Many-to-Many**: es. Article(N) →Category(M)

```
class Article(models.Model):
    ...
    category = models.ManyToManyField(Category)
```

L'opzione **Meta**:

```
class Article(models.Model):
    ...
    class Meta:
        unique_together = ("title", "pub_date")
        verbose_name = "article"
        verbose_name_plural = "articles"
```

Dettagli sulle query

- **Creare** una nuova istanza:

```
a = Article(title='titolo', content='contenuto articolo')
a.save()
```

- **Modificare** una istanza:

```
a.title = 'nuovo titolo'
a.save()
```

- Effettuare **query**:
  - Article.objects.**all**()
  - Article.objects.**get**(title="titolo articolo")
  - Article.objects.**filter**(pub_date__lte="2007-10-01")
  - Article.objects.**exclude**(pub_date__lte="2007-10-01")
  - Article.objects.all()**[:10]**

Internazionalizzazione

1. Utilizzare **ugettext_lazy**() nel codice python:
```
from django.utils.translation import ugettext_lazy as _
def index(request):
    return HttpResponse(_('Welcome to my blog'))
```
2. Utilizzare il blocco **trans** nei template:
{% **load i18n** %}
<title>**{% trans** "This is the title." **%}**</title>
3. Nelle view, impostare **request.LANGUAGE_CODE** e utilizzare **RequestContext**:
```
request.LANGUAGE_CODE = request.session['django_language']
render_to_response('blog/index.html', {...},
    context_instance=RequestContext(request))
```
4. Creare i file .po :~/mysite$ `make-messages.py -l it`
5. Modificare i .po e compilarli :~/mysite$ `compile-messages.py`

Integrazione con Apache

Modificare httpd.conf:

```
<Location "/mysite/">
    SetHandler python-program
    PythonHandler django.core.handlers.modpython
    SetEnv DJANGO_SETTINGS_MODULE mysite.settings
    PythonDebug On
    PythonPath "['/home/redclay/'] + sys.path"
</Location>

<Location "/media">
    SetHandler None
</Location>
```

Sommario

Twisted: caratteristiche principali

- Twisted è un framework per programmazione di rete
- Implementato a eventi (non bloccante)
- Multipiattaforma

Utilizza un loop (reactor) in cui avviene la gestione di tutti gli eventi

```
from twisted.internet import reactor
reactor.run()
```

Registrazione eventi

```
from twisted.internet import reactor
def callback():
    print 'called'
reactor.callLater(2, callback)
reactor.run()
```

Sommario

Deferred (1/2)

- I deferred permettono di aspettare la lettura/scrittura di dati dal sistema di I/O senza bloccare l'esecuzione del programma

Esempio: download di una pagina Web

```
from twisted.internet import reactor
from twisted.web.client import getPage

def print_data(data):
    print data
deferred = getPage('http://google.com')
deferred.addCallback(print_data)
reactor.run()
```

Deferred (2/2)

Creare un nuovo oggetto deferred

```
from twisted.internet import reactor, defer
def get_data():
    d = defer.Deferred()
    # registro un'operazione bloccante per l'I/O
    reactor.callLater(3, d.callback, 'data')
    return d
def print_data(data):
    print data
deferred = get_data()
deferred.addCallback(print_data)
reactor.run()
```

Importante

I deferred hanno senso solo se usati all'interno di un **reactor** (che rende asincrone le operazioni di I/O)

Deferred e Threads

In caso di funzioni che effettuano lunghi calcoli, è necessario l'uso dei thread

Usare i deferred con codice bloccante

```
from twisted.internet import reactor, threads
from time import sleep

def do_long_calculation():
    sleep(3)
    return 3
def print_result(x):
    print x
d = threads.deferToThread(do_long_calculation)
d.addCallback(print_result)
reactor.run()
```

Sommario

Protocol e Factory (1/2)

Un semplice client TCP di esempio

```
from twisted.internet.protocol import Protocol, ClientFactory

class ClientProtocol(Protocol):
    def connectionMade(self):
        print 'Connessione effettuata'
    def connectionLost(self, reason):
        print 'Connessione persa', reason
    def dataReceived(self, data):
        print data

class AppClientFactory(ClientFactory):
    protocol = ClientProtocol

from twisted.internet import reactor
reactor.connectTCP('127.0.0.1', 8000, AppClientFactory())
reactor.run()
```

Protocol e Factory (2/2)

Un semplice server TCP di esempio

```
from twisted.internet.protocol import Protocol, ServerFactory

class ServerProtocol(Protocol):
    def connectionMade(self):
        print 'Connessione effettuata'
        self.transport.write('data')
    def connectionLost(self, reason):
        print 'Connessione persa', reason

class AppServerFactory(ServerFactory):
    protocol = ServerProtocol

from twisted.internet import reactor
reactor.listenTCP(8000, AppServerFactory())
reactor.run()
```

Librerie incluse in Twisted

- **Core**: gestione asincrona degli eventi
- **Conch**: SSH e SFTP
- **Web**: HTTP
- **Mail**: SMTP, IMAP e POP
- **Names**: DNS
- **News**: NNTP
- **Words**: Chat e Instant Messaging (IRC)
- **Trial**: Unittest automated testing.
- **Lore**: generatore di documentazione con supporto HTML e LaTeX
- **Runner**: gestore di processi, include un server inetd

Riferimenti


http://python.org


http://www.djangoproject.com/, http://www.djangobook.com/


http://twistedmatrix.com

Licenza