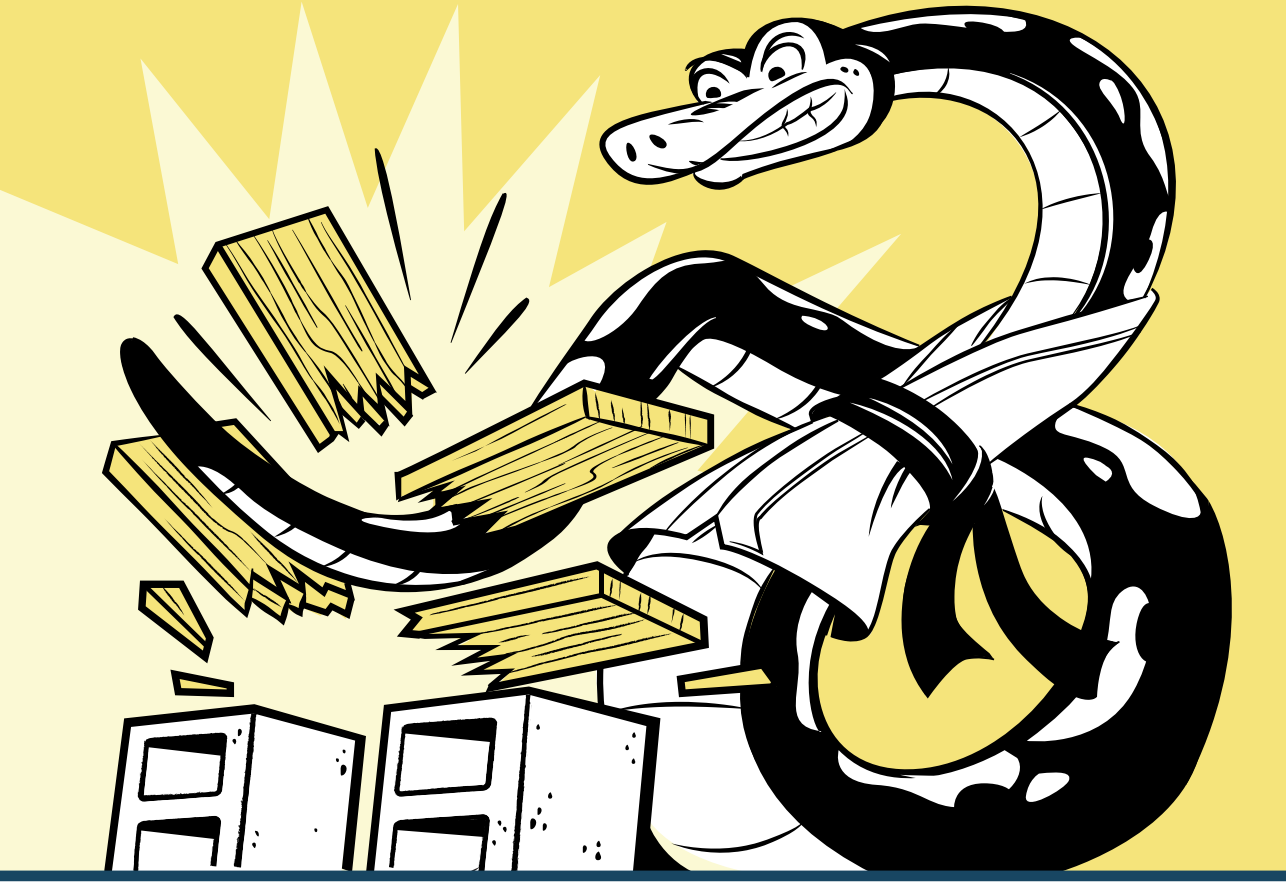
- Julien Danjou -

Python

Programmazione avanzata



Gestire timestamp e fusi orari, distribuzione, unit test >>

Metodi e decoratori, programmazione funzionale >>

Prestazioni e ottimizzazioni, scala e architettura >>

Gestione di database relazionali >>

no starch press

*pro DigitalLifeStyle

*pro
Digital*LifeStyle

Python

Programmazione avanzata

Julien Danjou

EDIZIONI
LSWR

Titolo originale: *Serious Python : Black Belt Advice on Deployment, Scalability, Testing, and More*
ISBN: 978-1-59327-878-6
Published by No Starch Press, Inc.
245 8th Street, San Francisco, CA 94103
www.nostarch.com
Cover Illustration: Josh Ellingson

Autore: Julien Danjou

Collana: *pro DigitalLifeStyle

Edizione italiana:
Python - Programmazione avanzata

Publisher: Marco Aleotti
Progetto grafico: Roberta Venturieri
Traduzione: Virginio B. Sala

ISBN: 978-88-6895-717-9

EDIZIONI LSWR

Via G. Spadolini, 7
20141 Milano (MI)
Tel. 02 881841
www.edizionilswr.it

Printed in Italy
Finito di stampare nel mese di aprile 2019 presso "Rotomail Italia" S.p.A., Vignate (MI)
(*) Edizioni Lswr è un marchio di La Tribuna Srl. La Tribuna Srl fa parte di LSWR GROUP.

Sommario

Ringraziamenti

Scrivere questo primo libro è stata una fatica tremenda. Ripensandoci, non avevo idea di quanto sarebbe stato folle questo viaggio ma non avevo idea neanche di quanto sarebbe stato gratificante.
Si dice che se si vuole andare veloci bisogna andare da soli, ma che se si vuole andare lontano bisogna andarci insieme. Questa è la quarta edizione del mio libro originale, e non ci sarei arrivato senza le persone che mi hanno aiutato lungo la strada. Questo è stato un lavoro di squadra e voglio ringraziare tutti quelli che vi hanno partecipato.
La maggior parte delle persone che ho intervistato mi hanno concesso il loro tempo e la loro fiducia senza pensarci due volte e devo a loro molto di quello di cui parlo nel libro: grazie a Doug Hellmann per i suoi ottimi consigli sulla costruzione di librerie, a Joshua Harlow per il buonumore e la sua conoscenza dei sistemi distribuiti, a Christophe de Vienne per la sua esperienza nella costruzione di framework, a Victor Stinner per la sua incredibile conoscenza di CPython, a Dimitri Fontaine per la sua competenza nei database, a Robert Collins per avere contribuito al testing, a Nick Coghlan per il suo lavoro nel dare una forma migliore a Python e a Paul Tagliamonte per il suo stupendo spirito hacker.
Grazie all'équipe della No Starch per aver lavorato con me per portare questo libro a un nuovo livello, in particolare a Liz Chadwick per le sue abilità redazionali, a Laurel Chun per avermi tenuto sulla strada giusta e a Mike Driscoll per le sue idee tecniche.
La mia gratitudine va anche alle comunità del software libero che hanno condiviso la loro conoscenza e mi hanno aiutato a crescere, in particolare alla comunità di Python, che è sempre stata accogliente ed entusiasta.

Introduzione

Se state leggendo queste pagine, è probabile che abbiate lavorato con Python già da un po' di tempo. Forse l'avete imparato da qualche tutorial, avete analizzato programmi scritti da altri o avete iniziato proprio da zero. In ogni caso, avete trovato la vostra strada per impararlo. È esattamente il modo in cui ho cominciato anch'io a conoscere Python, finché non ho iniziato a lavorare a grandi progetti open source una decina di anni fa.

È facile pensare di conoscere e capire Python quando si è scritto un primo programma. Il linguaggio è davvero semplice da afferrare. Ci vogliono anni, però, per padroneggiarlo e sviluppare una comprensione profonda dei suoi vantaggi e dei suoi punti deboli.

Quando ho iniziato, ho costruito le mie librerie e le mie applicazioni Python alla scala di un "progetto da garage". Le cose sono cambiate quando ho cominciato a lavorare con centinaia di sviluppatori a software su cui fanno affidamento migliaia di utenti. Per esempio, la piattaforma OpenStack (un progetto a cui contribuisco) rappresenta oltre 9 milioni di righe di codice Python, che nel loro complesso devono essere concise, efficienti e scalabili per le esigenze di qualsiasi applicazione di cloud computing richiedano i suoi utenti. Quando hai un progetto di queste dimensioni, cose come i test e la documentazione richiedono l'automazione, altrimenti non verranno fatte per niente.

Pensavo di sapere molto di Python prima di lavorare a progetti di questa scala, una scala che neanche potevo immaginare quando ho iniziato, ma ho imparato molto altro. Ho anche avuto la possibilità di incontrare alcuni fra i migliori hacker Python e di imparare da loro. Mi hanno insegnato di tutto, dall'architettura generale e dai principi di design a vari trucchi utili. In questo libro spero di condividere le cose più importanti che ho imparato, in modo che possiate costruire programmi Python migliori, e che possiate costruirli in modo più efficiente.

La prima versione in lingua inglese di questo libro è stata pubblicata nel 2014; ora siamo arrivati alla quarta edizione, con contenuti aggiornati e altri del tutto nuovi. Spero che vi piaccia.

Chi dovrebbe leggere questo libro e perché

Questo libro si rivolge agli sviluppatori Python che vogliono far fare un salto di qualità alle loro competenze. Vi troverete metodi e consigli che vi aiuteranno a ottenere il massimo da Python e a costruire programmi "a prova di futuro". Se state già lavorando su un progetto, potrete applicare direttamente le tecniche che analizzo qui, per migliorare il vostro codice attuale. Se dovete iniziare il vostro primo progetto, potrete crearne la struttura con le pratiche migliori.

Presenterò alcuni dei meccanismi interni di Python per darvi una comprensione migliore di come scrivere codice efficiente. Potrete acquisire una migliore conoscenza del funzionamento interno del linguaggio, che vi aiuterà a capire problemi o inefficienze.

Il libro fornisce anche soluzioni, collaudate sul campo, a problemi come i test, il porting, l'estensione della scala del codice, applicazioni e librerie Python. Questo vi aiuterà a evitare di compiere gli errori che altri hanno fatto e a scoprire strategie che contribuiranno alla manutenzione del vostro codice sul lungo periodo.

Come è organizzato questo libro

Questo libro non è stato progettato per essere letto obbligatoriamente dall'inizio alla fine. Potete saltare liberamente alle parti che vi interessano di più o che sono rilevanti per il vostro lavoro. In tutto il libro, troverete molti consigli e molti suggerimenti pratici. Questo è in breve il contenuto dei vari capitoli.

- Il **Capitolo 1** offre una guida a che cosa tenere in considerazione prima di intraprendere un progetto, con consigli sulla strutturazione del progetto, la numerazione delle versioni, l'impostazione di controlli automatizzati degli errori e altro ancora. Alla fine riporta un'intervista a Joshua Harlow.
- Il **Capitolo 2** introduce i moduli, le librerie e i framework Python e parla un po' di come funzionano internamente. Vi troverete una guida all'uso del modulo `sys`, a come ottenere di più dal gestore di package `pip`, alla scelta del framework più adatto e all'uso di librerie standard ed esterne. Alla fine, un'intervista a Doug Hellmann.
- Il **Capitolo 3** fornisce consigli sulla documentazione dei progetti e la gestione delle API mentre il progetto evolve anche dopo la pubblicazione. Troverete consigli specifici sull'uso di Sphinx per automatizzare alcune attività di documentazione. Infine, il capitolo si chiude con un'intervista a Christophe de Vienne.
- Il **Capitolo 4** tratta l'annoso problema dei fusi orari e di come gestirli al meglio nei programmi utilizzando oggetti `datetime` e `tzinfo`.
- Il **Capitolo 5** vi aiuterà a portare il vostro software agli utenti, con una guida alla distribuzione. Parleremo di packaging, di standard di distribuzione, delle librerie

`distutils` e `setuptools` e di come scoprire facilmente le funzionalità dinamiche in un package con i punti d'ingresso. Alla fine, un'intervista a Nick Coghlan.

- Il **Capitolo 6** offre consigli sugli unit test, con suggerimenti relativi alle buone pratiche e tutorial specifici sull'automazione degli unit test con `pytest`. Vedremo anche l'uso degli ambienti virtuali per aumentare l'isolamento dei test. L'intervista finale è a Robert Collins.
- Il **Capitolo 7** parla approfonditamente di metodi e decoratori. È uno sguardo all'uso di Python per la programmazione funzionale, con consigli su come e quando usare i decoratori e come creare decoratori *per* decoratori. Approfondiremo anche i metodi statici, di classe e astratti e vedremo come mescolarli per ottenere un programma più robusto.
- Il **Capitolo 8** presenta ulteriori "trucchi" di programmazione funzionale che si possono implementare in Python. In questo capitolo si parla di generatori, di comprensione di liste, di funzioni funzionali e di strumenti comuni per implementarle, e dell'utile libreria `functools`.
- Il **Capitolo 9** getta uno sguardo all'interno del linguaggio stesso e analizza l'albero sintattico astratto (AST) che costituisce la struttura interna di Python. Vedremo anche come estendere `flake8` per lavorare con l'AST e introdurre controlli automatici più sofisticati nei programmi. Il capitolo si conclude con un'intervista a Paul Tagliamonte.
- Il **Capitolo 10** è una guida all'ottimizzazione delle prestazioni mediante l'uso di strutture di dati appropriate, la definizione efficiente di funzioni e l'applicazione dell'analisi dinamica delle prestazioni per identificare i colli di bottiglia nel codice. Parleremo anche di memoizzazione e di riduzione degli sprechi con le copie di dati. Troverete anche un'intervista a Victor Stinner.
- Il **Capitolo 11** affronta il difficile tema del multithreading; parleremo fra l'altro di come e quando usare il multithreading piuttosto che il multiprocessing e se usare un'architettura orientata agli eventi oppure orientata ai servizi per creare programmi scalabili.
- Il **Capitolo 12** riguarda i database relazionali. Vedremo come funzionano e come usare PostgreSQL per gestire effettivamente e inviare in stream i dati. Alla fine, un'intervista a Dimitri Fontaine.
- Infine, il **Capitolo 13** offre alcuni consigli su vari argomenti: come rendere il codice compatibile sia con Python 2 sia con Python 3, come creare codice funzionale simile a quello Lisp, come usare i gestori di contesto e come ridurre la ripetizione con la libreria `attr`.

1

Avvio del progetto

In questo primo capitolo, esamineremo alcuni aspetti dell'avvio di un progetto e vedremo a che cosa bisogna pensare prima di iniziare, per esempio quale versione di Python utilizzare, come strutturare i moduli, come numerare efficacemente le versioni del software e come mettere in atto le migliori pratiche di codifica con il controllo automatico degli errori.

Versioni di Python

Prima di iniziare un progetto, dovete decidere quale versione o quali versioni di Python supporterà. Non è una decisione semplice come invece potrebbe sembrare. Non è un segreto che Python supporta numerose versioni contemporaneamente. Ogni versione secondaria dell'interprete ha il supporto di correzione dei bug per 18 mesi e il supporto di sicurezza per anni. Per esempio, Python 3.7, che è stato rilasciato il 27 giugno 2018, verrà supportato fino al rilascio di Python 3.8, previsto intorno a ottobre 2019. Intorno a dicembre 2019 ci sarà l'ultima release con correzione dei bug per Python 3.7, in previsione che tutti passino a Python 3.8. Ogni nuova versione di Python introduce nuove funzionalità e ne "depreca" altre più vecchie. La Figura 1.1 illustra questo tipo di calendario.
Oltre a questo, va preso in considerazione il problema del rapporto fra Python 2 e Python3. Chi lavora con piattaforme (molto) vecchie può avere bisogno ancora del supporto per Python 2, perché Python 3 non è disponibile su quelle piattaforme, ma la regola empirica è: lasciate perdere Python 2, se potete.

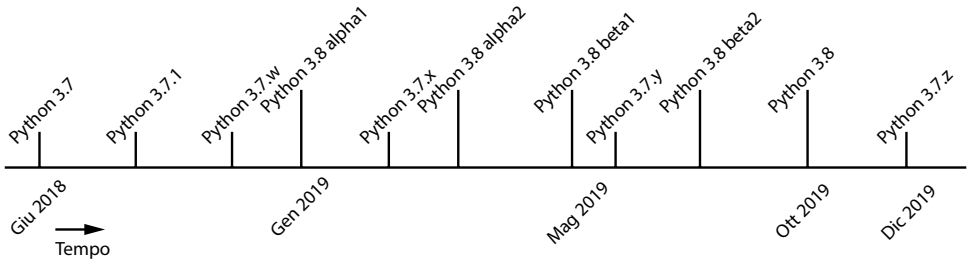


Figura 1.1 - La cronologia dei rilasci di Python.

Ecco un modo veloce per stabilire di quale versione avete bisogno.

- Le versioni 2.6 e precedenti sono ormai obsolete, perciò non consiglio di preoccuparvi di supportarle. Se intendete, per qualsiasi motivo, supportare queste versioni più vecchie, fate attenzione perché sarà difficile fare in modo che il vostro programma supporti anche Python 3.x. Detto questo, è ancora possibile incontrare Python 2.6 su qualche sistema più vecchio: in tal caso, mi spiace.
- La 2.7 è e resterà l'ultima versione di Python 2.x. Oggi ogni sistema è fondamentalmente in grado di eseguire, in un modo o nell'altro, Python 3, perciò, se non vi interessa l'archeologia, non dovete preoccuparvi di supportare Python 2.7 nei nuovi programmi. Quella versione non sarà più supportata dopo il 2020, perciò l'ultima cosa che vorrete è costruire nuovo software sulla sua base.
- La versione 3.7 è la più recente, nel ramo Python 3, al momento in cui scrivo, ed è quella a cui dovete mirare. Tuttavia, se il vostro sistema operativo è dotato della versione 3.6 (la maggior parte dei sistemi operativi, tranne Windows, è dotato di serie di una versione 3.6 o successiva), assicuratevi che la vostra applicazione lavori anche con la 3.6.

Delle tecniche per scrivere programmi che supportano sia Python 2.7 sia Python 3.x parleremo nel Capitolo 13.
Notate che questo libro è stato scritto pensando a Python 3.

Predisposizione del progetto

L'avvio di un nuovo progetto è sempre un po' un rompicapo. Non sapete ancora bene come sarà strutturato, perciò potreste non sapere come organizzare i vostri file. Tuttavia, sapendo quali sono le buone pratiche, capirete con quale struttura di base iniziare. Qui vi darò qualche consiglio su cosa fare e cosa no per predisporre il progetto.

Che cosa fare

Come prima cosa, considerate la struttura del vostro progetto, che deve essere molto semplice. Usate package e gerarchia in modo saggio: navigare una gerarchia profonda può essere un incubo, una gerarchia piatta tende a diventare gonfia.

Poi, evitate l'errore comune di conservare gli unit test all'esterno della directory di package. Questi test devono essere inclusi in un subpackage del vostro software, così che non vengano accidentalmente installati automaticamente come un modulo *tests* di massimo livello da `setuptools` (o da qualche altra libreria di packaging). Se li collocate in un subpackage, sarete sicuri che potranno essere installati e usati da altri package, così che gli utenti possano costruire i propri unit test.

La Figura 1.2 illustra come deve essere fatta una gerarchia di file standard.

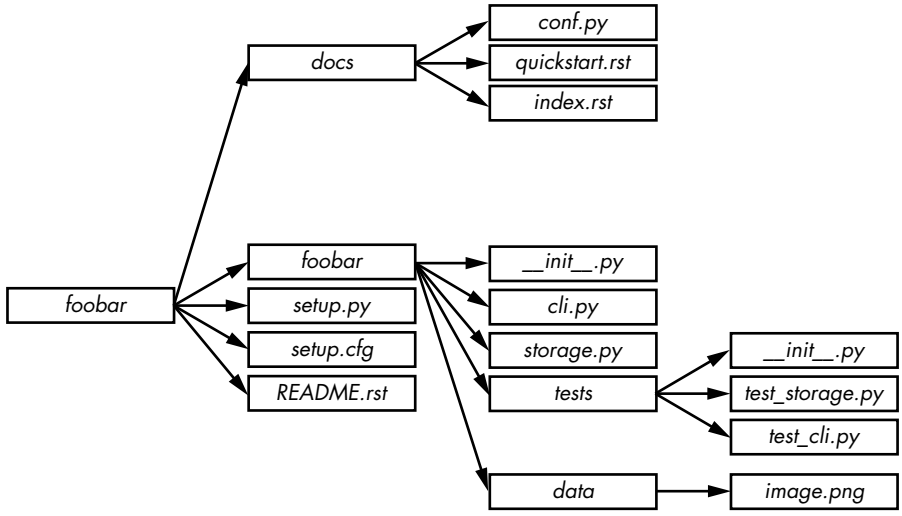


Figura 1.2 - Una directory di package standard.

Il nome standard per uno script di installazione di Python è *setup.py*. È accompagnato dal suo *setup.cfg*, che deve contenere i dettagli della configurazione dello script di installazione. Quando viene eseguito, *setup.py* installa il vostro package utilizzando le utility di distribuzione Python.

Potete fornire informazioni importanti per gli utenti in file *README.rst* (o *README.txt*, o con qualsiasi altro nome vi piaccia). Infine, la directory *docs* deve contenere la documentazione del package in formato *reStructuredText*, che sarà utilizzato da Sphinx (vedi il Capitolo 3).

I package spesso dovranno fornire ulteriori dati che il software dovrà usare, come immagini, script di shell e così via. Purtroppo, non esiste uno standard universalmente

accettato per la posizione in cui devono essere conservati questi file, perciò collocateli dove ha più senso per il vostro progetto, in base alla loro funzione. Per esempio, i modelli di applicazioni web possono andare in una directory *templates* sotto la directory *radice* del package.
Si trovano spesso anche queste directory di massimo livello:

- *etc* per i file di configurazione campione;
- *tools* per gli script di shell o altri tool correlati;
- *bin* per gli script binari che avete scritto e che saranno installati da *setup.py*.

Che cosa non fare

C'è un particolare aspetto di design che incontro spesso in strutture di progetto che non sono state ben pensate: qualche sviluppatore crea file o moduli sulla base del tipo di codice che ospiteranno. Per esempio, crea file *functions.py* o *exceptions.py*. Questa è una impostazione *terribile* e non aiuta nella navigazione del codice. Quando legge una base di codice, lo sviluppatore immagina che un'area funzionale di un programma sia confinata in un particolare file. L'organizzazione del codice non ha nulla da guadagnare da questa impostazione, che costringe chi legge a saltare da un file all'altro senza alcuna buona ragione.
Organizzate il vostro codice in base alle *funzionalità*, non ai tipi.
È una cattiva idea anche creare una directory di modulo che contenga solo un file *__init__.py*, perché è un annidamento non necessario. Per esempio, non dovete creare una directory *hooks* al cui interno si trovi un unico file *hooks/__init__.py*; *hooks.py* è sufficiente. Se create una directory, deve contenere vari altri file Python che appartengono alla categoria che la directory rappresenta. Costruire una gerarchia con livelli non necessari è fonte di confusione.
Dovete stare molto attenti anche al codice che inserite nel file *__init__.py*. Questo file verrà chiamato ed eseguito la prima volta che viene caricato un modulo contenuto nella directory. Se si mettono in un file *__init__.py* le cose sbagliate, è possibile che si verifichino effetti collaterali indesiderati. In effetti, i file *__init__.py* di solito devono essere vuoti, a meno che non sappiate quello che fate. Non tentate però di eliminare file *__init__.py*, altrimenti non riuscirete a importare il vostro modulo Python: perché la directory sia considerata un sottomodulo, Python richiede la presenza di un file *__init__.py*.

Numeri di versione

Le versioni del software devono essere identificate, in modo che gli utenti sappiano qual è la più recente. Per ogni progetto, gli utenti devono essere in grado di organizzare il calendario dell'evoluzione del codice. Esistono infiniti modi di organizzare i numeri

di versione, ma PEP 440 introduce un formato di versione che ogni package Python, e idealmente qualsiasi applicazione, deve seguire in modo che altri programmi e altri package possano identificare facilmente e in modo affidabile di quali versioni del vostro package abbiano bisogno.
PEP 440 definisce questo formato di espressione regolare per il numero di versione:

```
N[.N]+[{a|b|c|rc}N][.postN][.devN]
```

Questo consente la numerazione standard, del tipo `1.2` o `1.2.3`. Ci sono alcuni altri dettagli che è bene notare.

- La versione `1.2` è equivalente a `1.2.0`, `1.3.4` è equivalente a `1.3.4.0` e così via.
- Le versioni che corrispondono a `N[.N]+` sono considerate release *finali*.
- Versioni basate sulla data, come `2013.06.22`, sono considerate non valide. I tool automatizzati pensati per il rilevamento dei numeri di versione in formato PEP 440 solleveranno un errore se identificano un numero di versione maggiore o uguale a `1980`.
- I componenti finali possono usare anche il formato seguente:
  - `N[.N]+aN` (per esempio, `1.2a1`) denota una release alfa; è una versione che può essere instabile e priva di alcune funzionalità;
  - `N[.N]+bN` (per esempio, `2.3.1b2`) denota una release beta, una versione che può avere tutte le funzionalità, ma presentare ancora bug;
  - `N[.N]+cN` or `N[.N]+rcN` (per esempio, `0.4rc1`) denota una (release) candidate. Questa è una versione che potrà essere rilasciata come prodotto finale, a meno che non emergano bug significativi. I suffissi `rc` e `c` hanno lo stesso significato, ma se vengono usati entrambi, le release `rc` sono considerate più recenti delle release `c`.
- Possono essere utilizzati anche i suffissi seguenti:
  - il suffisso `.postN` (per esempio, `1.4.post2`) indica una post-release. Le postrelease normalmente sono utilizzate per risolvere piccoli errori nel processo di pubblicazione, per esempio errori nelle note di rilascio. Non dovete usare il suffisso `.postN` se rilasciate una versione che risolve dei bug; in quel caso, invece, incrementate il numero di versione secondario;
  - il suffisso `.devN` (per esempio, `2.3.4.dev3`) indica una release di sviluppo. Indica una prerelease della versione che specifica: per esempio, `2.3.4.dev3` indica la terza versione di sviluppo della release `2.3.4`, precedente a qualsiasi release alfa, beta, candidata o finale. Questo suffisso è scoraggiato perché è più difficile da analizzare per gli essere umani.

Questo schema dovrebbe essere sufficiente per la maggior parte dei casi comuni.

NOTA

Forse avrete sentito parlare di *Semantic Versioning*, che fornisce proprie direttive per i numeri di versione. Questa specifica ha alcune parti in comune con PEP 440, ma purtroppo le due non sono del tutto compatibili. Per esempio, la raccomandazione di Semantic Versioning per le prerelease usa uno schema come `1.0.0-alpha+001` che non è conforme a PEP 440.

Molte piattaforme di *sistemi distribuiti di controllo delle versioni* (DVCS) come Git e Mercurial possono generare numeri di versione con uno hash identificativo (per Git, vedi `git describe`). Purtroppo questo sistema non è compatibile con lo schema definito da PEP 440: per citare solo un aspetto, gli hash identificativi non sono ordinabili.

Stile di codifica e verifiche automatizzate

Lo stile di codifica è argomento delicato, ma ne dobbiamo parlare prima di entrare più in profondità in Python. A differenza di molti linguaggi di programmazione, Python usa i *rientri* (*indentation*) per definire i blocchi. Questo offre una soluzione semplice alla vecchia domanda "Dove devo mettere le parentesi graffe?", ma ne introduce una nuova: "Come devo fare i rientri?" Questa è stata una delle prime domande sollevate nella comunità, perciò il popolo di Python, nella sua grande saggezza, ha formulato la PEP 8: *Style Guide for Python Code* (*https://www.python.org/dev/peps/pep-0008/*), che definisce lo stile standard per il codice Python. Ecco in sintesi le direttive principali.

- Usate quattro spazi per livello di rientro.
- Limitate tutte le righe a un massimo di 79 caratteri.
- Separate le definizioni di funzioni e classi al livello più alto con due righe vuote.
- Codificate i file in ASCII o UTF-8.
- Usate una importazione di modulo per enunciato `import` e per riga. Collocate gli enunciati di importazione in testa al file, dopo i commenti e le docstring, raggruppati prima per standard, poi per terze parti e infine per import di librerie locali.
- Non usate spazi bianchi estranei fra parentesi tonde, quadre o graffe, o prima delle virgole.
- Scrivete i nomi di classe in "camel case" (per esempio, `CamelCase`), mettete come suffisso alle eccezioni `Error` (se applicabile), e date alle funzioni nomi in minuscole con parole e segni di underscore (per esempio, `separated_by_underscores`). Usate uno underscore iniziale per gli attributi o i metodi `_private`.

Queste direttive in realtà non sono difficili da seguire, e hanno molto senso. La maggior parte di quanti programmano in Python non hanno problemi a rispettarle.

Tuttavia *errare humanum est*, ed è ancora noioso riguardare il proprio codice per essere sicuri che sia conforme alle direttive PEP 8. Per fortuna, esiste un tool `pep8` (lo si trova all'indirizzo *https://pypi.org/project/pep8/*) che può controllare automaticamente qualsiasi file Python. Installate `pep8` con `pip`, poi potete usarlo su un file in questo modo:

```
$ pep8 hello.py
hello.py:4:1: E302 expected 2 blank lines, found 1
$ echo $?
1
```

Qui uso `pep8` sul mio file *hello.py*, e l'output indica quali righe e colonne non sono conformi a PEP 8 e mi comunica qualsiasi problema nel codice, qui alla riga 4 e colonna 1. Violazioni di enunciati *MUST* nella specifica sono indicati come *errori* e i relativi codici di errore iniziano con una *E*. Problemi secondari sono indicati come *warning* e i loro codici di errore iniziano con una *W*. Il codice a tre cifre che segue quella prima lettera indica il tipo esatto di errore o di warning.
La cifra delle centinaia indica la categoria generale di un codice di errore: per esempio, gli errori che iniziano con `E2` indicano problemi di spazi bianchi, quelli che iniziano con `E3` indicano problemi di righe vuote, i warning che iniziano con `W6` indicano l'uso di funzionalità deprecate. Questi codici sono tutti elencati nella documentazione di `pep8` (*https://pep8.readthedocs.io/*).

Tool per identificare gli errori di stile

La comunità discute ancora se sia una buona pratica validare codice rispetto a PEP 8, che non fa parte della Standard Library. Il mio consiglio è di prendere in considerazione l'esecuzione di un tool di convalida PEP 8 sul vostro codice sorgente con regolarità. Potete farlo facilmente integrandolo nel vostro sistema di integrazione continua. Questa impostazione può sembrare un po' estrema, ma è un buon modo per essere sicuri di continuare a rispettare le direttive PEP 8 sul lungo termine. Parleremo nel paragrafo "Usare `virtualenv` con `tox`" a pagina 95 di come si possa integrare `pep8` con `tox` per automatizzare queste verifiche.
La maggior parte dei progetti open source rende possibile la conformità a PEP 8 mediante verifiche automatiche. L'uso di queste verifiche automatiche sin dall'inizio del progetto può frustrare chi è alle prime armi, ma garantisce anche che la base di codice abbia lo stesso aspetto in ogni parte del progetto. Questo è molto importante per un progetto di qualsiasi dimensione, su cui lavorino più sviluppatori con idee diverse, per esempio, sull'ordine dello spazio bianco. Sapete che cosa intendo.
È anche possibile impostare il proprio codice perché ignori certi tipi di errori e di warning, utilizzando l'opzione `--ignore`, nel modo seguente:

```
$ pep8 --ignore=E3 hello.py
$ echo $?
0
```

Così tutti gli errori di codice E3 nel mio file *hello.py* verranno ignorati. L'opzione `--ignore` consente di ignorare effettivamente parti delle specifiche PEP 8 che non volete seguire. Se eseguite `pep8` su una base di codice esistente, vi consente anche di ignorare certi tipi di problemi, in modo da concentrarvi sulla risoluzione dei problemi una categoria alla volta.

NOTA

Se scrivete codice C per Python (per esempio, dei moduli), lo standard PEP 7 descrive lo stile di codifica che dovete seguire.

Tool per identificare errori di codifica

Python ha anche tool che controllano se esistono errori effettivi di codifica, anziché errori di stile. Ecco alcuni esempi degni di nota:

- *Pyflakes* (*https://launchpad.net/pyflakes/*): estendibile via plugin;
- *Pylint* (*https://pypi.org/project/pylint/*): verifica la conformità a PEP 8 mentre esegue controlli sugli errori del codice per default; può essere esteso via plugin.

Questi tool fanno tutti uso dell'analisi statica, cioè effettuano il parsing del codice e lo analizzano, invece di eseguirlo direttamente.
Se scegliete di usare Pyflakes, notate che non verifica di per sé la conformità con PEP 8, perciò dovrete usare il secondo tool `pep8` per coprire entrambi.
Per semplificare le cose, Python ha un progetto con il nome `flake8` (*https://pypi.org/project/flake8/*) che combina `pyflakes` e `pep8` in un unico comando. Aggiunge anche alcune nuove funzionalità interessanti: per esempio, può saltare le verifiche su righe che contengono un `# noqa` e può essere esteso via plugin.
Sono disponibii molti plugin per `flake8` che si possono usare così come sono. Per esempio, l'installazione di *flake8-import-order* (con `pip install flake8-import-order`) estenderà `flake8` in modo che controlli anche se i vostri enunciati `import` sono ordinati alfabeticamente nel vostro codice sorgente. Sì, ci sono progetti che lo richiedono.
Nella maggior parte dei progetti open source, `flake8` viene ampiamente usato per la verifica dello stile di codifica. Alcuni grandi progetti open source hanno addirittura scritto i loro plugin per `flake8`, aggiungendo verifiche per errori come usi errati di `except`, problemi di portabiità Python 2/3, stile di import, formattazioni errate di stringhe, possibili problemi di localizzazione e altro ancora.

Se avviate un nuovo progetto, vi consiglio di usare uno di questi tool per la verifica automatica della qualità e dello stile del codice. Se avete una base di codice che non implementava la verifica automatica, un buon metodo è lanciare il tool scelto con la maggior parte dei warning disabiitati e risolvere i problemi una categoria alla volta. Anche se nessuno di questi tool può essere perfetto per il vostro progetto o le vostre preferenze, `flake8` è un buon modo per migliorare la qualità del codice e garantirgli una maggiore durata.

NOTA

Molti text editor, fra cui anche il famoso GNU Emacs e vim, hanno plugin (come Flycheck) che possono eseguire tool come `pep8` o `flake8` direttamente nel buffer del codice, ed evidenziare interattivamente le parti del codice che non sono conformi a PEP 8. Questo è un modo comodo per risolvere la maggior parte degli errori di stile mentre si scrive del codice.

Nel Capitolo 9 parleremo di come estendere questo toolset con il nostro plugin per verificare la correttezza delle dichiarazioni di metodo.

Joshua Harlow su Python

Joshua Harlow è uno sviluppatore Python. È stato una delle guide tecniche del team OpenStack di Yahoo! fra il 2012 e il 2016 e ora lavora a GoDaddy. È autore di numerose librerie Python, fra cui *Taskflow*, *automaton* e *Zake*.

Che cosa ti ha portato a usare Python?

Ho cominciato a programmare in Python 2.3 o 2.4 ancora intorno al 2004 durante uno stage alla IBM vicino a Poughkeepsie, New York (gran parte dei miei parenti sono di quella parte dello stato di New York). Non ricordo più esattamente che cosa stessi facendo, ma riguardava wxPython e codice Python su cui stavano lavorando per automatizzare qualche sistema.

Dopo quello stage sono tornato a studiare, ho frequentato i corsi post laurea al Rochester Institute of Technology, e sono finito a lavorare a Yahoo!.

Alla fine sono entrato nel team del CTO, dove io e alcuni altri avevamo il compito di stabilire quale piattaforma cloud open source usare. Abbiamo deciso per OpenStack, che è scritta quasi completamente in Python.

Che cosa ami, e che cosa non ti piace, di Python?

Alcune delle cose che amo (l'elenco non è esaustivo).

- La sua semplicita: Python è davvero facile per chi è alle prime armi, ma riesce a tenere vivo l'interesse anche degli sviluppatori esperti.

- Il controllo dello stile: leggere più avanti nel tempo codice che hai scritto in precedenza è una parte considerevole dello sviluppo di software e avere una coerenza che può essere fatta valere da tool come `flake8`, `pep8` e Pylint è davvero di grande aiuto.
- La possibilità di scegliere stili di programmazione diversi e di mescolarli come risulta più opportuno.

Alcune delle cose che non mi piacciono (l'elenco non è esaustivo).

- La transizione un po' dolorosa fra Python 2 e 3 (la versione 3.6 ha risolto la maggior parte dei problemi, però).
- I lambda sono troppo semplici, devono diventare più potenti.
- La mancanza di un decente installatore di package. Penso che si debba lavorare ancora un po' su `pip`, per esempio sviluppare un vero risolutore di dipendenze.
- Il global interpreter lock (GIL) e il fatto che ce ne sia bisogno. Mi rattrista molto... [parleremo di GIL nel Capitolo 11].
- La mancanza di supporto nativo per il multithreading; al momento è necessario aggiungere un modello `asyncio` esplicito.
- La divisione della comunità di Python; questo riguarda soprattutto la divisione fra CPython e PyPy (e altre varianti).

Lavori a debtcollector, un modulo Python per gestire i warning di deprecazione. Com'è il processo di avvio di una nuova libreria?

La semplicità, di cui ho parlato prima, rende davvero facile preparare una nuova libreria e pubblicarla in modo che altri possano usarla. Dato che quel codice arriva da una delle altre librerie su cui lavoro (taskflow[1]) è stato relativamente facile trapiantare ed estendere il codice senza dovermi preoccupare del fatto che la API fosse progettata male. Sono molto felice che altri (all'interno della comunità OpenStack o all'esterno) abbiano trovato modo di usarla, e spero che la libreria cresca in modo da trattare altri stili di schemi di deprecazione che altre librerie (e applicazioni?) trovano utili.

Che cosa manca a Python, secondo te?

Python potrebbe avere prestazioni migliori con una compilazione just-in-time (JIT). La miaggior parte dei linguaggi più recenti (come Rust, Node.js che usa il motore JavaScript di Chrome V8, e altri ancora) hanno molte delle capacità di Python, ma hanno anche la compilazione JIT. Sarebbe davvero una bella cosa se anche CPython potesse

1 Persone che contribuiscano a questo progetto sono sempre le benvenute. Venite su IRC e partecipate a *irc://chat.freenode.net/openstack-state-management*.

essere compilato JIT in modo che Python potesse competere, sul piano delle prestazioni, con questi linguaggi più recenti.

Python ha bisogno anche di un robusto insieme di pattern di concorrenza; non solo `asyncio` di basso livello e stili threading di pattern, ma anche concetti di alto livello che contribuiscano a creare applicazioni che funzionino in modo performante su scala maggiore. La libreria `goless` porta in Python alcuni concetti di Go, che fornisce un modello di concorrenza incorporato. Sono convinto che questi pattern di livello più alto debbano essere disponibili come pattern di prima classe incorporati nella Standard Library e debbano essere mantenuti in modo che gli sviluppatori possano usarli quando lo ritengono opportuno. Senza questi, non vedo come Python possa competere con altri linguaggi che li mettono a disposizione.

Alla prossima, continuate a scrivere codice e siate felici!