ELM

PER PYTHONISTI

Avventure di un pythonista nel mondo dei linguaggi staticamente tipati.

Matteo Bertini

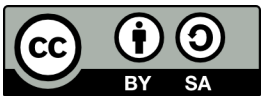
Elm per Pythonisti

Avventure di un pythonista nel mondo dei linguaggi staticamente tipati.

Matteo Bertini

Questo libro è in vendita presso http://leanpub.com/elm-per-pythonisti

Questa versione è stata pubblicata il 2016-08-03



A mia moglie Elisa ed ai miei figli.

Indice

Hug + Elm = fun

Prendi una libreria strettamente Python 3, sì, hai letto bene, la "versione di sviluppo" di Python [1]:

HUG: embrace the APIs of the future

ed un linguaggio funzionale compilato verso JavaScript e cosa ottieni?

[1] non è mia, l'ho sentita in un podcast dal "babbo" di Docker.

Elm lang

Sicuramente una ventata di freschezza e tanto tanto type checking.

Python è un linguaggio agile, è facile prototipare, ma con un po' di pratica è anche possibile creare progetti che sfidano gli anni [2], scegliendo i propri limiti invece di trovarseli imposti dal linguaggio. Questa libertà ha però un prezzo, in particolare i refactoring sono sempre un rischio perché il compilatore non fa praticamente niente più che trovare gli errori di sintassi e tutto il resto avviene a runtime. Per questo è obbligatorio avere una buona copertura di test ed è sempre buona prassi

[2] il progetto in cui lavoro, un CAD scritto in Python e C++, ha da poco compiuto 10 anni e supera le 200.000 linee di codice Python.

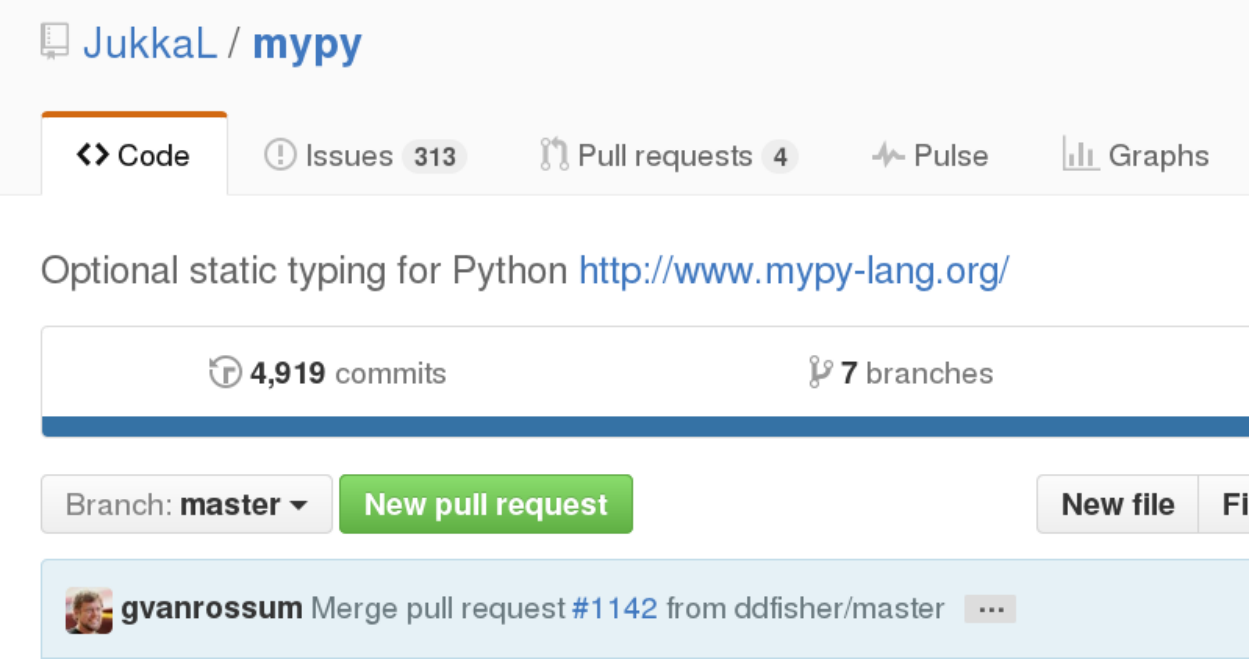
"blindare" la parte di codice da modificare con ulteriori test prima di affrontare qualsiasi modifica non banale.

Avendo giocato un po' con OCaml e Rust a volte mi manca l'aiuto del compilatore quando scrivo in Python, spesso davvero l'errore che era sfuggito nel codice è poco più che un errore di sintassi.

Ad esempio quante volte nello scrivere un algoritmo ho deciso che un `set` era una struttura dati più adatta di una `list`? Tante, e quante le volte che nel rinominare `data.append(x)` in `data.add(x)` me ne sono scordato uno in un remoto `if` che ha dato errore solo grazie ai test? Tante, e quante invece l'errore è saltato fuori solo usando il programma? Meno per fortuna, ma capita.

Ma non dovrebbe, nel 2016 mi aspetto di più da un linguaggio... ed infatti si può avere di più!

Python dalla versione 3.0 accetta delle annotazioni ad argomenti e valori di ritorno delle funzioni [3], queste annotazioni sono libere ma in Python 3.5 è stato introdotto il modulo `typing` e sono state standardizzate le annotazioni di tipo. Ad oggi non esiste un type checker integrato in Python, ma il progetto mypy[4], da cui è partito tutto, oggi vede come ultimo committer il nostro Guido!



Guido commits on mypy

Questo mi da ottime speranze per il futuro, perché ad oggi *mypy*[5] non mi è stato molto utile, dato che solleva una valanga di errori nel controllare le dipendenze del mio mini progetto.

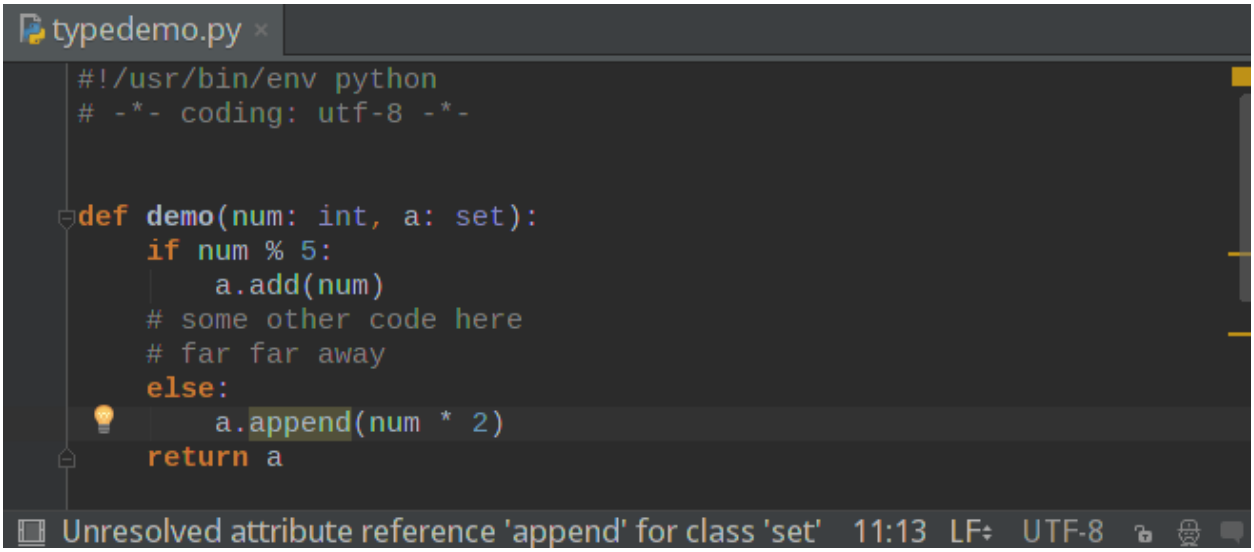
Ma con un piccolo file di esempio invece l'errore che ci aspettiamo compare subito:

[3] http://python-future.org/func_annotations.html
[4] http://www.mypy-lang.org
[5] `pip3 install mypy-lang` e non `mypy`

```python
def demo(num: int, a: set):
    if num % 5:
        a.add(num)
    # some other code here.
    # ...
    # far far away.
    else:
        a.append(num * 2)
    return a
```

```
$ mypy typedemo.py
typedemo.py: note: In function "demo":
typedemo.py:12: error: Set[Any] has no attribute "append"
```

Per chi preferisce una soluzione più integrata, una IDE come PyCharm già integra un type checker ed in tempo record ha cominciato a supportare la neonata sintassi:



PyCharm typedemo.py

Quindi possiamo sperare che presto anche in Python tanti errori banali potranno essere individuati a "check-time" invece che durante i test o peggio durante l'uso.

Ma in realtà Python in questo tipo di controlli è un novellino, c'è un linguaggio che ha fatto degli errori di compilazione per essere umani il suo cavallo di battaglia. Elm[6] è un linguaggio funzionale che si compila in JavaScript, il "traduttore" è scritto in Haskell. Non temete, di fatto non vi capiterà spesso di doverlo compilare, perché viene distribuito come pacchetto binario per `npm`.

[6] http://elm-lang.org

Nato come libreria per scrivere giochi, si è evoluto negli ultimi anni come un linguaggio completo per la scrittura di fontend complessi, senza niente da invidiare a framework più famosi, dato che integra tutte le buzzword del momento: *virtual-dom*, *functional reactive programming*, *time-travelling debugger*, e la più importante *no runtime exceptions*.

Come è possibile non avere eccezioni a runtime? Per prima cosa non devono esistere i tipi nullabili, niente più `null` o `None` su cui chiamare metodi o da passare a funzioni che si aspettano un valore concreto, se una variabile può non avere un valore dovremo definirla come `Maybe` oppure `Optional` in altri linguaggi.

Un `Maybe <tipo a>` rappresenta una opzione tra `Just <valore di tipo a>` e `Nothing`. Questo da solo non è molto diverso da come facciamo spesso in python, cioè usare implicitamente `None` per codificare l'assenza di un valore. Però dato che in python questa modifica è implicita, non sapremo mai da una visione parziale del codice se il valore `None` è previsto come caso possibile oppure no.

Quindi per completare il puzzle è necessario un pattern matching esaustivo, Vediamo come definire un `Maybe` in Elm e come il compilatore ci aiuta nel gestire esaustivamente tutti i casi che potrebbero manifestarsi a runtime.

```
1  > var = Just 123
2  Just 123 : Maybe.Maybe number
```

Abbiamo definito la variabile `var` come un `Just <numero>`. `Just` non è altro che una delle due varianti del tipo `Maybe`, definito come:

```
1  type Maybe a = Nothing | Just a
```

Grazie all'inferenza di tipi, anche se in questo caso non abbiamo definito il tipo di `var`, questo viene dedotto dal letterale `123 : number` e quindi `var` assume il tipo `var : Maybe number`, in particolare questa istanza ha valore `Just 123`.

A questo punto per accedere al numero contenuto useremo il pattern matching:

```
1  > case var of\
2  | Just num -> num
```

ma in questo caso ci siamo scordati di gestire il caso `Nothing` in cui il valore non è presente (l'esempio è minimale, ma immaginate l'accesso per chiave ad un dizionario che può non contenere la chiave). Questo codice è incompleto, `num` non è definito se `var` è `Nothing`, e fino alla scorsa versione di Elm avremmo dovuto aspettare il momento dell'esecuzione per scoprire che un ramo del match non era coperto, ma dall'ultima versione il compilatore è diventato più "intelligente" e ci fa notare subito il problema.

```
1  ============================== ERRORS ==============================
2
3  -- MISSING PATTERNS ---------------------------- repl-temp-000.elm
4
5  This `case` does not have branches for all possibilities.
6
7  4|>  case var of
8  5|>  Just num -> num
9
10 You need to account for the following values:
11
12     Maybe.Nothing
13
14 Add a branch to cover this pattern!
```

Ma fa anche altro, se volete un assaggio vi consiglio di guardare il blog post Compilers as Assistants[7] dell'autore di Elm, Evan Czaplicki[8].

Con questi fondamenti (e se siete dei teorici dei linguaggi potrete elencare gli altri requisiti) e con un passaggio controllato per accedere al mondo *usafe* del JavaScript, abbiamo un linguaggio coerente per scrivere codice che girerà su una VM JavaScript senza scrivere mai un rigo di JavaScript.

Lo so, JavaScript non è un brutto linguaggio, non peggio di altri, ma non ho particolare interesse ad imparare un lunguaggio che assomiglia molto a Python come linguaggio dinamico e amichevolmente anarchico. Però se voglio fare qualcosa che sia compatibile col presente, *web 2.0*, *se-non-c'è-la-web-app-non-esiste* ho l'ottima opportunità di usare JavaScript solo come linguaggio target per un compilatore.

L'introduzione si è protratta fin troppo, passiamo all'implementazione.

Per questo esperimento mi sono preso la libertà di usare framework non mainstream: - per le API ho scelto HUG[9], - a sua volta HUG usa falcon[10], un framework WSGI minimale, - invece come ORM ho scelto peewee[11].

Il backend della nostra app «Guess the number» ha 2 endpoint dinamici:

- `POST /play` per iniziare una nuova partita, con `id` univoco e `token` random associato,
- `PUT /game/{gameid}/guess/{guess}` per cercare di indovinare un numero.

Oltre a questi il backend serve il file statici `index.html` e l'app JS, `elm.js`.

[7] http://elm-lang.org/blog/compilers-as-assistants
[8] https://twitter.com/czaplic
[9] https://github.com/timothycrosley/hug
[10] http://www.falconframework.org/
[11] http://docs.peewee-orm.com/

Potete usare un workspace cloud9 per fare le prove, clonando quello che ho configurato nello scrivere questo post: ide.c9.io/naufraghi/guess[12]

[12] https://ide.c9.io/naufraghi/guess