لماذا_بايثون_ ؟

وثائق أعجوبة
http://docs.ojuba.org

رابط الوثيقة : http://www.ojuba.org/wiki/docs:لماذا_بايثون
ساهم في تحرير هذه الوثيقة :
مؤيد السعدي - أحمد محمد عبد الرحمن - علي الشمري - أشرف علي خلف - الشريف أحمد - أحمد فيصل - الصادق الشريدي

أول تحرير بواسطة مؤيد السعدي بتاريخ 2008/04/24 02:20
آخر تحرير بواسطة مؤيد السعدي بتاريخ 2009/09/14 06:51
تم تصدير هذه الوثيقة بتاريخ : 2010/01/02 09:49

تنويه : تمثل هذه الوثيقة تصديرا لنص على موقع أعجوبة، ولكن رغم ذلك
لا يتحمل الموقع أية مسئولية قانونية عن صحة أو خطا ما يرد فيها.

يسمح لك نسخ أو توزيع أو تعديل هذا المستند
وفق شروط الرخصة الحرة المحددة
حقوق النسخ محفوظة ©

جدول المحتويات

1. لماذا بايثون ؟.....4
1.1. حول هذه الوثيقة.....4
1.2. مقدمة.....4
1.3. تكشف بيرل.....4
1.4. إعادة النظر في بايثون.....4
1.5. حفراً إلى الأعماق.....4
1.6. الخلاصة.....4

~~ODT~~

1. لماذا بايثون ؟

1.1. حول هذه الوثيقة

هذا مقال مترجم

- تأليف: إريك ريموند Eric Raymond المعروف اختصاراً ESR وهو مؤيد ناشط لنظام لينكس ومؤلف The Cathedral & The Bazaar يمكن مراسلته على esr@thyrsus.com
- المقال الأصلي نشرته عدة مواقع منها
  - http://www.python.org/about/success/esr/
  - www.linuxjournal.com/article/3882

1.2. مقدمة

نظرتي الأولى للغة بايثون كانت مصادفة، ولم أحب ما رأيته في ذاك الوقت كثيراً. وكان ذلك في بداية 1997، وقتها كان كتاب برمجة بايثون Programming Python لمؤلفه مارك لوتز Mark Lutz من أورالي قد صدر تواً. وكتب اورالي تأتي عتبتي من حين لآخر ينتقيها لي من بين إصداراتهم متبرع غامض داخل المؤسسة من خلال عملية عشوائية يأست من فهمها.

أحدها (كتاب من أورالي) كانت برمجة بايثون Programming Python. وقد وجدت هذا ممتعاً بطريقة ما، حيث أني أجمع لغات الحاسوب. فأنا أعرف أكثر من دزينتين من اللغات عامة الأغراض، وأكتب عدداً من المصنفات compilers والمفسرات interpreters للمتعة، وقد صممت أي عدد من اللغات خاصة الأهداف وأشكال الإرقام markup formalisms بنفسي. آخر مشاريعي المكتملة أثناء كتابة هذه المقالة، لغة خاصة الأهداف اسمها SNG للتعامل مع ملفات صور PNG. إذا كنت مهتماً انظر للصفحة الرئيسية www.catb.org/~esr/sng كذلك كتبت بعض لغات البرمجة عامة الأغراض الشاذة في معرض الحوسبة الرجعية www.catb.org/retro

لقد سمعت بالفعل عن بايثون بما يكفي أن أعرف أنها -كما يسمى في وقتنا الحاضر- "لغة سكربتية" ، ولغة تفسيرية تتمتع بإدارة جيدة للذاكرة مدمجة بها،وحسن إدارة الموارد للاستدعاء والتعاون مع البرامج الأخرى. لذا فإنني غصت في برمجه بايثون ،وسؤال واحد يدور فى خاطري : ما الذي لدى هذه اللغة وليس لدى بيرل مثله ؟!

بيرل بالطبع ؛ هي عملاق برمجة السكربت الحديثة ، وقد حلت إلى حد كبير- محل لغة برمجة شل سكربت ، كخيار لمديري النظام . وذلك -جزئيا- بفضل مكتبة يونيكس الشاملة ونداءات النظام ، وكذلك بفضل المجموعة الهائلة من وحدات بيرل البرمجية والتي بنيت من قبل مجتمع بيرل البالغ النشاط . وعموما تشير التقديرات إلى كونها لغة الواجهة المشتركة لبوابة الشبكة-common- **CGI** gateway interface و التي تقف وراء حوالي 80% من المحتوى الحي على الشبكة . "لاري وول" مؤلف لغة بيرل يعتبر بحق واحدا من أهم قادة مجتمع المصادر المفتوحة ،وغالبا ما يأتي في المرتبة الثالثة بعد لينوس تورفالدز وريتشارد ستولمان ضمن عظماء الهاكرز "المقدسين" في الوقت الحاضر .

في ذلك الوقت ، كنت أستخدم بيرل لعدد من المشروعات الصغيرة. وقد وجدت انها قوية جدا ، حتى

لو كان التركيب وبعض الجوانب الاخرى من اللغة تبدو عرضة للخطأ الفادح اذا لم تستخدم بعناية.و بدا لي تماما ان بايثون مثل تل آخر علي تسلقه مثل لغات البرمجة الأخرى ، وذلك كما قرأت ، بحثت أولا عما يبدو انها مستقلة عن بيرل .

وعلى الفور تعثرت بأول سمة من سمات بايثون الغريبة والتي يمكن للجميع ملاحظتها : وهي المسافات الفارغة "الإزاحة"(indentation) وهي في الواقع ذات أهمية كبيرة في تراكيب اللغة. واللغة لا تناظر سي أو بيرل في استخدام الحاصرة في التراكيب ،وبدلا من ذلك ، فإنها تغير في الإزاحة وتنظيم مجموعات التصريحات . و مثلما وقع لمعظم الهاكرز عند إدراكهم الأول لهذه الحقيقة ، لقد شعرت بالاشمئزاز .

وأنا بالكاد كبير بما يكفي أن يكون لدي برمجيات من فورتران ترجع لعدة أشهر في السبعينات. معظم الهاكرز لم يوجدوا في هذه الأيام ، ولكن يبدو أن ثقافتنا تميل -الى حد ما- إلى الإبقاء على ذكرى جميلة لهذا الطراز القديم والكريه ذي المنظر الثابت في مجال لغات البرمجة. والواقع أن مصطلح "الشكل الحر" ، والمستخدم في ذلك الوقت لوصف أحدث تركيب لغوي رمزي المنحى في باسكال و سي،أصبح -تقريبا-في طي النسيان ،وقد تم تصميم جميع اللغات -أو أغلبها- بهذا الأسلوب لعدة عقود الآن.على أية حال؛ من الصعب إلقاء اللوم على أي شخص على رؤية هذه الميزة في بايثون ، وكرد فعل أولي وغير متوقع لفكرة الخوض في تبخير كومة من روث الديناصور.

وهذا بالتأكيد ما شعرت به. وقد استخلصت ما تبقى من زُبدة هذه اللغة دونما اهتمام كبير. ولم أر كثيرا مما يوصَي به في بايثون ، ماعدا ما يبدو من كونها أنظف في التركيب اللغوي من بيرل ، والتسهيلات التي تقدمها للعناصر الأساسية في الواجهة الرسومية للمستخدم مثل الأزرار والقوائم، والتي -بأمانة-تبدو جميلة.

أعدت الكتب إلى الرّف ملصقاً ملاحظة على عقلي بأن علي أن أكتب بعد الكود لبرنامج ذو واجهة رسومية GUI بلغة باثون في وقت ما، لا لشيء إلا لأثبت أني فهمت اللغة. لكني لم أكن مقتنعاً أن رأيت منها يمكن أن ينافس بيرل.

1.3. تكشف بيرل

ولقد تآمرت كثير من الأمور الأخرى لإبقاء تلك الفكرة في ذيل قائمة أولوياتي لعدة أشهر. وقد كان ما تبقى من عام 1997 حافلا بالأحداث بالنسبة لي ، ومن ضمن الأمور الأخرى أنه العام الذي كتبت ونشرت فيه النص الاصلي لكتاب"الكاتدرائية والسوق". وقد وجدت وقتا لكتابة عدة برامج Perl ، ومن ضمنها برنامجان كبيرا الحجم ومعقدان . احدهما:Keeper، وهو برنامج مساعد ما يزال يستخدم في السيطرة على الطلبات القادمة لأرشيف برنامج metalab . وهو يقوم بتوليد صفحات الويب التي تراها في metalab.unc.edu/pub/Linux/!INDEX.html. والبرنامج الآخر ، anthologize ،ويستخدم لتوليد PostScript تلقائيا في الإصدار السادس الخاص بلينكس من مشروع أرشيف وثائق لينكس من howtos. وكلا البرنامجين متوفر على metalab كتابة هذه البرامج ترك فيّ تدريجياً عدم رضى عن بيرل. كلما كبر حجم المشروع فإن المنغصات في بيرل تتضخم إلى مشاكل جدية ومستمرة. الصياغة التي بدت سلسة عند مئة سطر بدأت تصبح سياجاً منيعاً من الأشواك عند الالف. "هناك أكثر من طريقة للقيام بها"[1] أعارنا النكهة والتعبير بالقليل. لكنه صعب المحافظة على نسق واحد على قاعدة كبيرة من الكود. والعديد من المزايا التي تم إرقاعها فيما بعد في بيرل لمواجهة التحكمات المعقدة التي تتطلبها البرامج الكبيرة (الكائنات، تحديد النطاق scoping و وجملة use strict لزيادة الصرامة ...) الذي يعطيني شعور بأنه هش ومتلكف وبني دون أساس.

These problems combined to make large volumes of Perl code seem unreasonably difficult to read and grasp as a whole after only a few days' absence. Also, I found I was spending more and more time wrestling with artifacts of the language rather than my application problems. And, most

1 شعار بيرل

damning of all, the resulting code was ugly—this matters. Ugly programs are like ugly suspension bridges: they're much more liable to collapse than pretty ones, because the way humans (especially engineer-humans) perceive beauty is intimately related to our ability to process and understand complexity. A language that makes it hard to write elegant code makes it hard to write good code.

With a baseline of two dozen languages under my belt, I could detect all the telltale signs of a language design that had been pushed to the edge of its functional envelope. By mid-1997, I was thinking “there has to be a better way” and began casting about for a more elegant scripting language.

One course I did not consider was going back to C as a default language. The days when it made sense to do your own memory management in a new program are long over, outside of a few specialty areas like kernel hacking, scientific computing and 3-D graphics—places where you absolutely must get maximum speed and tight control of memory usage, because you need to push the hardware as hard as possible.

For most other situations, accepting the debugging overhead of buffer overruns, pointer-aliasing problems, malloc/free memory leaks and all the other associated ills is just crazy on today's machines. Far better to trade a few cycles and a few kilobytes of memory for the overhead of a scripting language's memory manager and economize on far more valuable human time. Indeed, the advantages of this strategy are precisely what has driven the explosive growth of Perl since the mid-1990s.

I flirted with Tcl, only to discover quickly that it scales up even more poorly than Perl. Old LISPer that I am, I also looked at various current dialects of Lisp and Scheme—but, as is historically usual for Lisp, lots of clever design was rendered almost useless by scanty or nonexistent documentation, incomplete access to POSIX/UNIX facilities, and a small but nevertheless deeply fragmented user community. Perl's popularity is not an accident; most of its competitors are either worse than Perl for large projects or somehow nowhere near as useful as their theoretically superior designs ought to make them.

1.4. إعادة النظر في بايثون

نظرتى الثانية للبايثون كان تقريبا صدفة كالصدفة الأولى . فى أكتوبر من عام 1997 , سلسلة من الأسئلة على fetchmail [1] من الأصدقاء عبر البريد الإلكترونى هذه القائمة من الرسائل فسرت أن المستخدمين النهائيين يعانون من زيادة الخلل فى توليد ملفات الإعدادت لـ fetchmail الخاص بى . هذا الملف يستخدم نظام اليونكس الكلاسيكى المجانى للإعراب [2] , لكنه يمكن أن يكون صعبا عندما يملك المستخدم حسابات مثل POP3 و IMAP على مواقع متعددة. وكمثال انظر اللائحة الاولى LISTING1 حيث تشاهد نسخة مبسطة من إعداداته عندي

LISTING 1: fetchmail Configuration File

```
set postmaster "esr"
set daemon 300
```

1 هو برنامج معقد الإعدادات يستخدم لجلب البريد لمعالجته و عمل قوائم بريدية
2 UNIX free-format syntax

```
poll imap.ccil.org with proto IMAP and options no dns
    aka snark.thyrsus.com locke.ccil.org ccil.org
      user esr there is esr here options fetchall dropstatus warnings 3600
poll imap.netaxs.com with proto IMAP
      user "esr" there is esr here options dropstatus warnings 3600
skip imap.21cn.com with proto IMAP
      user esr here is tranxww there options fetchall
skip pop.tems.com with proto POP3:
      user esr here is ed there options fetchall
skip mail.frequentis.com with proto IMAP:
      user esr here is imaptest there with options fetchall
```

قررت أن أهاجم المشكلة بكتابة محرر إعدادات صديق للمستخدم، fetchmailconf. وكان الهدف التصميمي من fetchmailconf واضحا: أن يخفي تماما رموز التحكم بالملف خلف واجهة رسومية أنيقة مزودة بأزرار التحديد والشرائط المنزلقة والنماذج التي يمكن ملؤها.

وفكرة تضمين هذا كله في Perl لم أتحمس لها مطلقا.. فقد رأيت كود الواجهة الرسومية في Perl، وكان خليطا متنافرا بين Perl و Tcl. وبدأ أبشع بكثير من كود Perl الخالص الذي كتبته..وعند هذه النقطة تذكرت الجزء الذي كنت وضعته بأكثر من ستة شهور قبل ذلك. قد تكون هذه فرصة سانحة للحصول على بعض الخبرة العملية في بايثون.

Of course, this brought me face to face once again with Python's pons asinorum, the significance of whitespace. This time, however, I charged ahead and roughed out some code for a handful of sample GUI elements. Oddly enough, Python's use of whitespace stopped feeling unnatural after about twenty minutes. I just indented code, pretty much as I would have done in a C program anyway, and it worked.

That was my first surprise. My second came a couple of hours into the project, when I noticed (allowing for pauses needed to look up new features in Programming Python) I was generating working code nearly as fast as I could type. When I realized this, I was quite startled. An important measure of effort in coding is the frequency with which you write something that doesn't actually match your mental representation of the problem, and have to backtrack on realizing that what you just typed won't actually tell the language to do what you're thinking. An important measure of good language design is how rapidly the percentage of missteps of this kind falls as you gain experience with the language.

When you're writing working code nearly as fast as you can type and your misstep rate is near zero, it generally means you've achieved mastery of the language. But that didn't make sense, because it was still day one and I was regularly pausing to look up new language and library features!

This was my first clue that, in Python, I was actually dealing with an exceptionally good design. Most languages have so much friction and awkwardness built into their design that you learn most of their feature set long before your misstep rate drops anywhere near zero. Python was the first general-purpose language I'd ever used that reversed this process.

Not that it took me very long to learn the feature set. I wrote a working, usable fetchmailconf, with GUI, in six working days, of which perhaps the equivalent of two days were spent learning Python itself. This reflects another useful property

of the language: it is compact–you can hold its entire feature set (and at least a concept index of its libraries) in your head. C is a famously compact language. Perl is notoriously not; one of the things the notion “There's more than one way to do it!” costs Perl is the possibility of compactness.

1.5. حفراً إلى الأعماق

But my most dramatic moment of discovery lay ahead. My design had a problem: I could easily generate configuration files from the user's GUI actions, but editing them was a much harder problem. Or, rather, reading them into an editable form was a problem.

The parser for fetchmail's configuration file syntax is rather elaborate. It's actually written in YACC and Lex, two classic UNIX tools for generating language-parsing code in C. In order for fetchmailconf to be able to edit existing configuration files, I thought it would have to replicate that elaborate parser in Python. I was very reluctant to do this, partly because of the amount of work involved and partly because I wasn't sure how to ascertain that two parsers in two different languages accept the same. The last thing I needed was the extra labor of keeping the two parsers in synchronization as the configuration language evolved!

This problem stumped me for a while. Then I had an inspiration: I'd let fetchmailconf use fetchmail's own parser! I added a –configdump option to fetchmail that would parse .fetchmailrc and dump the result to standard output in the format of a Python initializer. For the file above, the result would look roughly like Listing 2 (to save space, some data not relevant to the example is omitted).

Listing 2: fetchmailrc

```
fetchmailrc = {
    'poll_interval':300,
    "logfile":None,
    "postmaster":"esr",
    'bouncemail':TRUE,
    "properties":None,
    'invisible':FALSE,
    'syslog':FALSE,
    # List of server entries begins here
    'servers': [
    # Entry for site `imap.ccil.org' begins:
    {
        "pollname":"imap.ccil.org",
        'active':TRUE,
        "via":None,
        "protocol":"IMAP",
        'port':0,
        'timeout':300,
        'dns':FALSE,
        "aka":["snark.thyrsus.com", "locke.ccil.org", "ccil.org"],
        'users': [
        {
            "remote":"esr",
```

```
            "password":"Malvern",
            'localnames':["esr"],
            'fetchall':TRUE,
            'keep':FALSE,
            'flush':FALSE,
            "mda":None,
            'limit':0,
            'warnings':3600,
        }
        ,         ]
    }
    ,
    # Entry  for  site  `imap.netaxs.com'  begins:
    {
        "pollname":"imap.netaxs.com",
        'active':TRUE,
        "via":None,
        "protocol":"IMAP",
        'port':0,
        'timeout':300,
        'dns':TRUE,
        "aka":None,
        'users':  [
        {
            "remote":"esr",
            "password":"d0wnthere",
            'localnames':["esr"],
            'fetchall':FALSE,
            'keep':FALSE,
            'flush':FALSE,
            "mda":None,
            'limit':0,
            'warnings':3600,
        }
        ,         ]
    }
    ,
    # Entry  for  site  `imap.21cn.com'  begins:
    {
        "pollname":"imap.21cn.com",
        'active':FALSE,
        "via":None,
        "protocol":"IMAP",
        'port':0,
        'timeout':300,
        'dns':TRUE,
        "aka":None,
        'users':  [
        {
            "remote":"tranxww",
            "password":None,
            'localnames':["esr"],
            'fetchall':TRUE,
            'keep':FALSE,
            'flush':FALSE,
            "mda":None,
            'limit':0,
```

```
            'warnings':3600,
        }
        ,         ]
    }
    ,
    # Entry for site `pop.tems.com' begins:
    {
        "pollname":"pop.tems.com",
        'active':FALSE,
        "via":None,
        "protocol":"POP3",
        'port':0,
        'timeout':300,
        'dns':TRUE,
        'uidl':FALSE,
        "aka":None,
        'users':  [
        {
            "remote":"ed",
            "password":None,
            'localnames':["esr"],
            'fetchall':TRUE,
            'keep':FALSE,
            'flush':FALSE,
            "mda":None,
            'limit':0,
            'warnings':3600,
        }
        ,         ]
    }
    ,
    # Entry for site `mail.frequentis.com' begins:
    {
        "pollname":"mail.frequentis.com",
        'active':FALSE,
        "via":None,
        "protocol":"IMAP",
        'port':0,
        'timeout':300,
        'dns':TRUE,
        "aka":None,
        'users':  [
        {
            "remote":"imaptest",
            "password":None,
            'localnames':["esr"],
            'fetchall':TRUE,
            'keep':FALSE,
            'flush':FALSE,
            "mda":None,
            'limit':0,
            'warnings':3600,
        }
        ,         ]
    }
    ]
}
```

Python could then evaluate the fetchmail –configdump output and have the configuration available as the value of the variable “fetchmail”.

This wasn't quite the last step in the dance. What I really wanted wasn't just for fetchmailconf to have the existing configuration, but to turn it into a linked tree of live objects. There would be three kinds of objects in this tree: Configuration (the top-level object representing the entire configuration), Site (representing one of the sites to be polled) and User (representing user data attached to a site). The example file describes five site objects, each with one user object attached to it.

I had already designed and written the three object classes (that's what took four days, most of it spent getting the layout of the widgets just right). Each had a method that caused it to pop up a GUI edit panel to modify its instance data. My last remaining problem was somehow to transform the dead data in this Python initializer into live objects.

I considered writing code that would explicitly know about the structure of all three classes and use that knowledge to grovel through the initializer creating matching objects, but rejected that idea because new class members were likely to be added over time as the configuration language grew new features. If I wrote the object-creation code in the obvious way, it would be fragile and tend to fall out of sync when either the class definitions or the initializer structure changed.

What I really wanted was code that would analyze the shape and members of the initializer, query the class definitions themselves about their members, and then adjust itself to impedance-match the two sets.

This kind of thing is called metaclass hacking and is generally considered fearsomely esoteric—deep black magic. Most object-oriented languages don't support it at all; in those that do (Perl being one), it tends to be a complicated and fragile undertaking. I had been impressed by Python's low coefficient of friction so far, but here was a real test. How hard would I have to wrestle with the language to get it to do this? I knew from previous experience that the bout was likely to be painful, even assuming I won, but I dived into the book and read up on Python's metaclass facilities. The resulting function is shown in Listing 3, and the code that calls it is in Listing 4.

Listing 3: Metaclass Function

```
def copy_instance(toclass, fromdict):
# Initialize a class object of given type from a conformant dictionary.
    class_sig = toclass.__dict__.keys(); class_sig.sort()
    dict_keys = fromdict.keys(); dict_keys.sort()
    common = intersect(class_sig, dict_keys)
    if 'typemap' in class_sig:
    class_sig.remove('typemap')
    if tuple(class_sig) != tuple(dict_keys):
        print "Conformability error"
#       print "Class signature: " + `class_sig`
#       print "Dictionary keys: " + `dict_keys`
        print "Not matched in class signature: " + `setdiff(class_sig, common)`
```

```
        print "Not matched in dictionary keys: " + `setdiff(dict_keys, common)`
        sys.exit(1)
    else:
    for x in dict_keys:
        setattr(toclass, x, fromdict[x])
```

Listing 4: Code that Calls Metaclass Function

```
# The tricky part--initializing objects from the
#   configuration global
# `Configuration' is the top level of the object
#   tree we're going to mung
Configuration = Controls()
copy_instance(Configuration, configuration)
Configuration.servers = [];
for server in configuration[`servers']:
    Newsite = Server()
    copy_instance(Newsite, server)
    Configuration.servers.append(Newsite)
    Newsite.users = [];
    for user in server['users']:
        Newuser = User()
        copy_instance(Newuser, user)
        Newsite.users.append(Newuser)
```

That doesn't look too bad for deep black magic, does it? Thirty-two lines, counting comments. Just from knowing what I've said about the class structure, the calling code is even readable. But the size of this code isn't the real shocker. Brace yourself: this code only took me about ninety minutes to write—and it worked correctly the first time I ran it.

1.6. الخلاصة

To say I was astonished would have been positively wallowing in understatement. It's remarkable enough when implementations of simple techniques work exactly as expected the first time; but my first metaclass hack in a new language, six days from a cold standing start? Even if we stipulate that I am a fairly talented hacker, this is an amazing testament to Python's clarity and elegance of design.

There was simply no way I could have pulled off a coup like this in Perl, even with my vastly greater experience level in that language. It was at this point I realized I was probably leaving Perl behind.

This was my most dramatic Python moment. But, when all is said and done, it was just a clever hack. The long-term usefulness of a language comes not in its ability to support clever hacks, but from how well and how unobtrusively it supports the day-to-day work of programming. The day-to-day work of programming consists not of writing new programs, but mostly reading and modifying existing ones.

So the real punchline of the story is this: weeks and months after writing fetchmailconf, I could still read the fetchmailconf code and grok what it was doing without serious mental effort. And the true reason I no longer write Perl

for anything but tiny projects is that was never true when I was writing large masses of Perl code. I fear the prospect of ever having to modify keeper or anthologize again—but fetchmailconf gives me no qualms at all.

Perl still has its uses. For tiny projects (100 lines or fewer) that involve a lot of text pattern matching, I am still more likely to tinker up a Perl-regexp-based solution than to reach for Python. For good recent examples of such things, see the timeseries and growthplot scripts in the fetchmail distribution. Actually, these are much like the things Perl did in its original role as a sort of combination awk/sed/grep/sh, before it had functions and direct access to the operating system API. For anything larger or more complex, I have come to prefer the subtle virtues of Python—and I think you will, too.