From Python to Pythonic: Preliminary Results from Searching for Python idioms in GitHub

José Javier Merchante
Universidad Rey Juan Carlos
Fuenlabrada, Madrid, Spain
Email: jj.merchante@gmail.com

Gregorio Robles
GSyC/LibreSoft
Universidad Rey Juan Carlos
Fuenlabrada, Madrid, Spain
Email: grex@gsyc.urjc.es

I. Introduction

Every programming language has its culture and usual way to code a task; that's what programmers usually call *idioms* [2]. For an advanced programmer in a given language there is always a *better* way of accomplishing a task that is more suitable in that language (e.g., it improves its readability) instead of writing the implementation it replaces in the same way as in another language.

Python is a programming language that in the last years has grown a lot. For this language there are many tools that check the code against very common style conventions (such as the ones specified in the PEP-8[1]), but there is to our knowledge no tool that identifies what idioms a program contains, or that helps improving your Python code making it more idiomatic (commonly referred to as more *Pythonic* [6]). Even though no such tool exist, the Python community is concerned a lot about these issues and many books, articles, talks and references on how to make your Python code more Pythonic can be found.

Many Python books and web pages explain the language without including these idioms, and focus on explaining the language as it would be another programming language, but with Python syntax. As an example, the following is correct Python:

```
colors = ["blue", "red", "yellow"]

for i in range(len(colors)):
    print colors[i]
```

However, even if the code runs and works perfectly, there is a more *Pythonic* way of doing it:

```
colors = ["blue", "red", "yellow"]

for color in colors:
    print color
```

For these reasons, we think it would be a good idea to analyze Python idioms and create a tool that can help beginners and advanced programmers to make their Python code more legible, readable, and write the task the *right*, *Pythonic* way.

II. Methodology

For the implementation of this tool we have searched of the most important Python idioms in books and talks at PythonCon, the most important conference on Python. By this way, more that 100 Python idioms of various difficulty levels have been collected. This list contains idioms such as:

- Comprehensions
- Magic methods
- Lambda functions
- Decorators
- Collections structures
- Class methods
- Closures
- ...

In order to identify the idioms in Python source code, we look for tokens in Python code. For some idioms, this is straightforward (e.g., for the *with* keyword), while for others it is more complex (e.g., for list comprehensions, in particular those that contain Boolean logic). Then, we scan Python code in a project with a lexer called `Pygments`[2] for these idioms.

For each idiom found, we also retrieve and store meta-information from the versioning repository, such as its author (we therefore use `git blame`). That allows to obtain the Pythonic contribution of each collaborator to a GitHub project and in this way make statistics.

In order to find out the suitability of our tool and to perform a first study of the use of Pythonic elements in real Python code, we mined all projects with Python as main language from GitHub. We used GHTorrent [3] as data source for knowing the main programming language of a repository[3], of

[1]PEP 8 – Style Guide for Python Code: https://www.python.org/dev/peps/pep-0008/.

[2]http://pygments.org/

[3]We do only consider those repositories that are non-forks.

which 700,000 (out of over 15 million) were in Python. Out of these, we downloaded a sample of 70,000 projects.

Given the amount of data to be downloaded and analyzed, the analysis of the repositories took five weeks. Projects were downloaded, and then non-Python files in them were removed in order to store space. All in all, at the end of the process we had a total of 500 GB of Python files. These were analyzed with our tool. The output was redirected to a database and analyzed with Pandas [4].

III. Some preliminary results

In this section, we present some preliminary results.

Figure III shows the number of idioms per repository. This curve shows a powerlaw distribution with a long tail: many of the repository have few idioms, although more than 50% of the repositories have more than 40 idioms – the mean is 304.52. Noteworthy is the fact that around 12% of the repositories do not contain any of the Python idioms in their code.

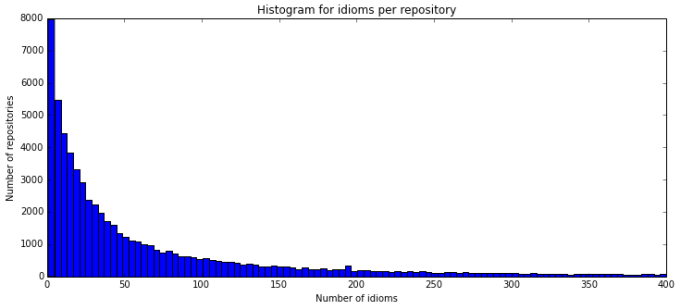

Fig. 1. Number of idioms per repository

Figure III displays the most used idioms by number of repositories, counting an idiom only once per repository. As we can be observed, decorators and list comprehensions are two of the most common Python idioms, although the most frequent ones are the use of named arguments in function calls and *docstrings* (attached comments in a specific format that serves as documentation for classes and methods). The `if __name__ == "__main__"` construct is also at least one time in a repository for over 80% of the cases.

Some of the most used and known Python idioms are *magic* methods (methods that are invoked when using a certain syntax, starting and finishing with __, also known as *dunder* from *double under*). These methods provide characteristics that, if implemented, are *transversal* to classes and thus provide a common, defined way for some functionality. An example of this are the __repr__, which returns a printable representation of an object, and __str__, which returns a string containing a nicely printable representation of an object, methods. Figure III gives the number of occurrences of each of the most frequent magic methods. This time, the total number of appearances, i.e., a magic method that appears N times in a repository will count N for that repository, is given. The most frequent one, `__init__` (the initializer method, used in Python in a constructor-ish nature), is omitted in the figure as it appears many more times than the rest.

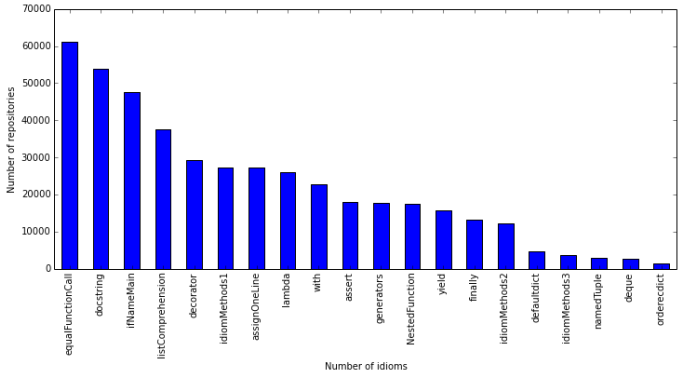

Fig. 2. Ranking of most frequent Python idioms. Idioms are counted only once per repository.

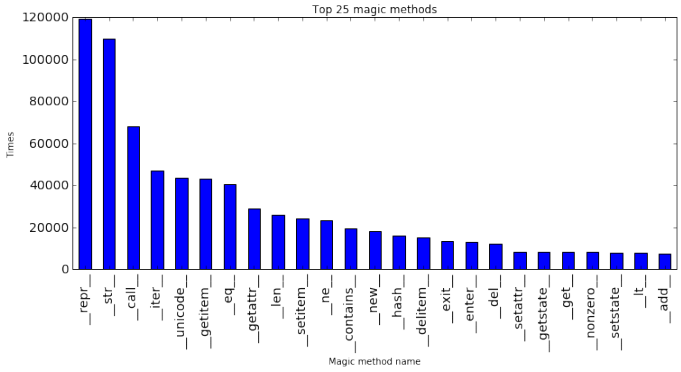

Fig. 3. Most common magic methods (__init__ not included).

As can be seen from Figure III, __repr__ and __str__ are the two most frequent magic methods. The next ones are: __call__ that implements a function call operator, __iter__ to create iterators, and __unicode__ which in Python 2 returns characters as __str__ really returns bytes[4].

IV. Future work

This paper shows work-in-progress in our quest for finding how idioms are used in Python. In the near future, we would like to do the following:

- Extend and assess the list of Python idioms. We would like to have a list of idioms as complete as possible. These should be evaluated by Python developers.
- There are idioms that are conceptually more difficult than others. We would like, again with the help of Python developers, see if we can classify the idioms by their complexity.

[4]In Python 3, the __str__ magic method returns characters as in Python's 2 __unicode__, and a new __bytes__() magic method exists.

- If idioms are *good* practices, we have noticed as well the existence of *anti-idioms* (similar to the patterns and anti-patterns idea [1]). We would like to identify them and see how often they are used.
- We would like to filter projects by their importance, first by omitting pet or student projects (for instance those that have a lifetime of less than 6 months) and second by giving a weight to projects by using data from the Python Package Index (PIP).
- We would like to create a web-tool to evaluate Python repositories on-the-fly for their idioms (and anti-idioms).
- We would like to create a web-tool that allows developers to evaluate their Python skills based on their contributions. Our idea would be to, once we know the GitHub username, scan for the idioms a developer has used and attending to their difficulty, etc., assign the developer a level of mastery in the Python language. This tool would be similar to the Dr. Scratch tool used by children to assess the development of their computational thinking skills [5].
- We would like to study how Python idioms get propagated. This has two perspectives: how do new Python idioms propagate, and how do developers learn them.

V. Acknowledgements

The work of Gregorio Robles has been funded in part by the Region of Madrid under project "eMadrid - Investigación y Desarrollo de tecnologías para el e-learning en la Comunidad de Madrid" (S2013/ICE-2715) and in part by the Spanish Government under project SobreVision (TIN2014-59400-R).

References

[1] W. H. Brown, R. C. Malveau, H. W. McCormick, and T. J. Mowbray. *AntiPatterns: refactoring software, architectures, and projects in crisis*. John Wiley & Sons, Inc., 1998.

[2] J. Coplien. Advanced c++ programming styles and idioms. In *Technology of Object-Oriented Languages and Systems, 1997. TOOLS 25, Proceedings*, pages 352–352. IEEE, 1997.

[3] G. Gousios and D. Spinellis. Ghtorrent: Github's data from a firehose. In *Mining software repositories (msr), 2012 9th ieee working conference on*, pages 12–21. IEEE, 2012.

[4] W. McKinney. *Python for data analysis: Data wrangling with Pandas, NumPy, and IPython*. O'Reilly Media, Inc., 2012.

[5] J. Moreno-Leon, G. Robles, and M. Roman-Gonzalez. Dr. scratch: Automatic analysis of scratch projects to assess and foster computational thinking. *RED. Revista de Educación a Distancia*, 15(46), 2015.

[6] G. Van Rossum et al. Python programming language. In *USENIX Annual Technical Conference*, volume 41, 2007.