When Importless Becomes Meaningful:

A Call for Tag-based Namespaces in Programming Languages

Tomas Tauber
The University of Hong Kong
ttauber@cs.hku.hk

Abstract

Traditional programming language namespaces evolved from filesystem structures. We describe different scenarios where this rigid code organization becomes a limiting factor. After that, we propose a more flexible code organization using tags. We then illustrate it on Python, including how we can convert existing code structures to the new tag-based one. Finally, we discuss our plans how to extend this work to statically typed languages in the future.

Categories and Subject Descriptors D.3.3 [*Programming Languages*]: Language Constructs and Features – Modules, packages

Keywords Modules; Packages; Tags; Namespaces; Python

1. Motivation

The way we do computer programming has not changed much over the last 40 years. Despite various improvements in development tools and new programming paradigms, we still store computer program source codes as static text files (except, for example, in JetBrains MPS[9]) in traditional hierarchical filesystems. This fact influences many design decisions in programming languages and their compilers.

One such decision is how we deal with naming collisions. Compilers scan all files on given local filesystem paths and complain if they find symbols with the same names. Programmers then need to rename these symbols, or make use of namespaces if their language supports this concept. In general, namespaces in programming languages have been more or less mimicking namespaces in hierarchical filesystems. Apart from a scoping role in language semantics and occasional meta-programming support, they do not differ much and suffer from similar problems:

- A program unit can only belong to one namespace. A class, a function or a constant can only belong to one module. A module can only belong to one package. In hierarchical namespaces, it may also belong to its parent packages, but it is still a restriction. A hierarchical code organization can be a suitable choice in a variety of cases and we do not argue against that. We rather argue that programmers or organizations should make this choice themselves.

SPLASH '14, October 20-24 2014, Portland, OR, USA.

ACM 978-1-4503-3208-8/14/10.
http://dx.doi.org/10.1145/2660252.2660257

- It assumes a traditional filesystem file as a base storage unit which may not be always meaningful. One could possibly create a file for each "program unit", but that may not be the best option. On the other hand, grouping different "program units" may not always make a semantic sense. Examples of that are "utility" or "commons" packages that tend to contain various pieces of different functionality. Reorganizing the code or temporarily grouping different parts of it in "virtual modules" may not be possible without using complex tools.
- Default querying, plain-text pattern matching, of text files is quite limiting. In filesystems, "desktop search engines" overcome this limitation. In programming languages, IDEs and various development tools allow better code searching or visualization. Even so, it is not an ideal situation and especially in dynamic languages, one can imagine how these tools would benefit from better querying support (e.g. more relevant code suggestions based on the surrounding code).

2. Problem

Beyond these inherent problems, we may encounter issues more relevant to software development:

- Programs contain semantic information that is being ignored, but may be useful for code organization and dependencies, querying, or visualization: information about versions, dependencies (e.g. a language version), authors, actual functionality, or data structures. They, however, are not standard parts of languages and may be inconsistent which is a reason why language runtimes ignore such information. Again, IDEs and other development tools may still make use of it.
- Portability of program units may be reduced. Even though individual units, such as functions, may be independent of each other, they may have inherent dependencies arising from their placement. For example, if one tries to copy a part of a code and paste it either in an interactive command line interface or in a different module, it will probably not work: 1) there might be naming collisions with existing identifiers in the current namespace, 2) there might be "unknown" identifiers (of unimported modules) even though an interpreter could find such unique identifiers on given search paths.
- Interactions among program units change over time and the structure may stop making sense. MacCormack, Rusnak and Baldwin [5] studied design evolution of a commercial software product over time. They found out that software architects' original choices did not match interactions and component structures in later versions. We cannot foresee such architectural changes in the future and refactoring may be fairly complex due to the rigid namespace structure and compatibility requirements.

- There are issues that the use of traditional namespaces will not resolve. For example, if we try to import two different versions of the same library, it will be impossible and we will need to use qualified names and disambiguate them using the filesystem paths. A similar issue is with the same version of a library targeting different versions of the language. We can resolve these issues using build systems and other tools, such as `virtualenv` in Python. Nonetheless, such solutions are rather ad-hoc workarounds due to the imposed namespace design restrictions.

To illustrate some of these issues, we consider the following simple example in Python:

```
from weather import get_data
from numpy import sum
  (... some code ...)
def avg_temp():
  data = get_data()
  avg = sum(data) / len(data)
  return avg
  (... some code ...)
```

If we try to move this function definition to a different module, there might be problems with name clashes or unimported names (if we do not copy the related import statements). And, for example, we cannot select a specific version of `numpy` when multiple versions exist on the search path.

3. Approach

The mentioned problems exist, because language designs do not address them and leave it to external tools. Newly proposed tag-based filesystems, such as TagFS [2] or hFAD [7], solve the inherent issues in operating systems. Inspired by them, we explore this idea in the context of programming language namespaces.

3.1 Concept

We can imagine it in a language that lacks an import (require, include etc.) statement and has an implicit "autoload" for all names from the search path. For dynamic languages, this is trivial if a name referenced in a code is unique (e.g. `len()`). It may not be correct, but we do not get many guarantees anyway and it stresses the importance of testing. Statically typed languages require a more careful treatment and we will address them in the future work. Now, let us consider a scenario with non-unique names. We first need to perform "name mangling" of the colliding identifiers and keep track of these symbols. The question is then how to disambiguate.

As shown below, we annotate different parts of code with tags in comments. These tags not only describe functionality, but also potential dependencies and implicitly guide loading by ranking the overlap of tags. For instance, `get_data()` may be defined in `weather` and `stock_exchange` libraries and they provide different functionality. Tags `#weather` and `#temperature` appear in the former one, hence the function loads from there.

For `sum()`, we may have multiple options: Python's standard library, our own definition or other packages, such as `numpy`. All `sum()` definitions provide more or less the same functionality and may contain the same set of tags (e.g. `#summation`). In this case, we need to be more specific and provide more tags, either on the function definition level or before the call. For our definition, for example, we could add `#author:NAME`. If there are no more tags, we can then either modify the original definition and decorate it with more tags, or if that is not desirable, we can retag it locally:

```
# #weather #average #temperature
# #@#('.../site-packages/numpy/', #ndarray)
def avg_temp():
  data = get_data()
  # #summation #ndarray #version:1.8.1
  avg = sum(data) / len(data)
  return avg
```

One advantage this code organization brings is self-documentation. Naturally, different people may give different tags to the same code, but as with existing code organizations, this issue should be addressed in coding conventions of particular projects. Other advantages are that it helps to address the mentioned issues: we gain a greater flexibility in code organization; we can index program units by their tags and by names they import; we can reorganize the code by generating temporary "workspace" virtual modules containing code that matches a query; program units carry all their dependencies locally; we may have different versions of the same library on the search path and disambiguate based on tags, such as `#version:1.0` or `#pythonversion:2.7`.

Such tag-based "importless" schemes can be retrofitted as preprocessors to most language runtimes of current dynamic languages. We demonstrate it on CPython 2.7.6.

3.2 Tagging

There are three levels at which we can tag our code:

1. **Top**: This level includes classes, functions or variables defined in a global scope. For classes and functions, everything in their scope has at least the surrounding set of tags. These annotations are not only guiding imports in that scope, but are also exposing how particular units can be referenced and used.
2. **Block**: Python does not have code blocks on their own, so we refer to control-flow constructs: `while`, `if-elif-else`, `for`, `try-except-finally`, and `with-as`. We can annotate a control flow block and all its statements may gain tags in addition to the ones defined at the surrounding top-level.
3. **Statement**: Finally, we can annotate individual statements on their preceding lines.

From the code organization perspective, we use the block- and statement-level tagging for isolated disambiguation that does not interfere with the rest of the code. We could, however, still use it to guide code searching – i.e. we can extract and index tags from all levels for a particular unit, but we may want to assign them with different weights.

3.3 Converting Existing Hierarchies

We adopt the following procedure to convert existing code structures:

- Given a source file, annotate each top-level unit with tags generated from its module name and all its parent packages. All imported modules and their parent packages generate tags that annotate a corresponding level, i.e. all top-level units (if in the beginning of the module) or the enclosing scope of a particular block (if it is a local import statement).
- For all references to names from the standard library, annotate their preceding lines with a tag `#__builtin__`.
- For all import statements, remove module prefixes of each reference and annotate its preceding line with the module name tag.
- For all import-as statements, generate retagging statements with paths to these modules and tags from aliases. Remove each alias prefix and annotate its preceding line with the corresponding tag.
- Tag any-module level executable code blocks with `#__main__`.
- Remove all import statements.

This approach is basic and possible future work can improve upon it by extracting keywords and annotations from docstrings and annotating corresponding parts of code with them.

3.4 Storing and Editing

For the ease of querying, we store top-level units and their tags in a relational database. We can achieve it using two tables (or three if a more normalized solution is desirable): the first table stores names, "description" (class, function, variable), mangled names, paths, and actual code; the second one references the first table and stores corresponding tags for each unit. We can then specify temporary file names and what code they should contain based on tags. A tool, running in the background, can continuously watch changes in these files (e.g. via *inotify*) and sync changes with the database by diffs: update or delete code, add or remove tags, update names and regenerate the mangled names, or add new entries.

3.5 Loading

The interpreter runs a code block specified by a name and tags. For the code execution, we essentially follow the reverse conversion process in pre-processing (using Python's `tokenize`) and regenerate original code organization: we infer explicit imports from tags and insert them to code along with possible module or qualified name prefixes. There may be program code that was added to temporary files and does not have a physical filesystem location. We dump all such units into a single temporary module under their mangled names. We then add import statements for this temporary module to files that are referencing these units and replace the names with the mangled names in them.

In order to support dynamic code (`exec`, `eval` or command-line interface), we need to modify Python's internals on initialization and add an exception hook (`sys.excepthook`) to handle *NameErrors*. When we catch a *NameError*, we query the database for a given name and try to resolve it; if we cannot resolve it, we raise an error and print all matching names along with their tags.

4. Evaluation Methodology

The primary hypothesis is that it is possible to use the more flexible tag-based namespaces instead of the traditional hierarchical namespaces in a dynamic language without sacrificing "code quality". The secondary hypothesis is that while using this code organization, one can still achieve a comparable performance.

We are going to test these hypotheses on a converted bundle of existing popular packages. In particular, we plan to convert packages from the Anaconda 1.9.2 collection. For the primary hypothesis, we compare cohesions of modules before the conversion and of corresponding virtual modules. We use the LCOM4 metric [3] on the module level (rather than the class level) to assess the cohesion.

For the secondary hypothesis, we utilize Python's `timeit` module to measure execution time. As a baseline, we first measure runtime performance of multiple runs of unit tests for each unconverted package. In order to gain consistent steady-state results, we execute 3 warm-up runs before running 10 measured runs, and force garbage collection before each run. We measure in this setting for each package's unit test collection in separate VM invocations. We will follow the same procedure with the tag-namespace aware interpreter executing on the converted packages.

5. Related Work

The closest to this work is keyword programming [4]. In keyword programming, users provide unordered keywords which are then translated into queries against APIs and search results generate detailed expressions in the given code context. In attribute-oriented programming [6], programmers mark parts of programs to indicate application- or domain-specific semantics and preprocessors generate more detailed programs for these annotated parts. Namespaces, however, remain the same as in the original language (Java).

Other generative and metaprogramming paradigms, such as intentional programming [1], allow programmers to create code in a higher level of abstraction. They also address the issues related to productivity by making program code more accessible for tools. These approaches often imply a completely new programming model which may not be easily retrofitted to existing languages.

UpgradeJ [8] was an extension to the Java programming language that focused on the problem of dynamic software updating (hotswapping). It provides an explicit upgrade mechanism and allows multiple versions of the same classes to co-exist in the same namespace by forcing class names to be annotated with a version number.

6. Future Work

After finishing the prototype for dynamic languages, we would like to extend this work to statically typed languages. We need to examine the semantics of tag-based namespaces in this scenario. And we may need to extend the current scheme: attaching weights to tags, different ways of combining them or defining relationships among them, such as different restrictions on which tags can coexist in the same annotation sets. Under this defined semantics, we will explore the possibility of type inference. In addition to it, we will look into how tag-based namespaces can be combined with existing programming paradigms – for instance, how tags should be exposed and propagated with respect to notions of ownership and inheritance in OOP.

References

[1] W. Aitken, B. Dickens, P. Kwiatkowski, O. D. Moor, D. Richter, and C. Simonyi. Transformation in intentional programming. *Proceedings. Fifth International Conference on Software Reuse (Cat. No.98TB100203)*, 1998. ISSN 1085-9098. .

[2] S. Bloehdorn, O. Görlitz, S. Schenk, M. Völkel, and F. I. Karlsruhe. Tagfs - tag semantics for hierarchical file systems. In *I-KNOW 06: Proceedings of the 6th International Conference on Knowledge Management*, pages 6–8, 2006. .

[3] M. Hitz and B. Montazeri. Measuring Coupling and Cohesion In Object-Oriented Systems. *Angewandte Informatik*, 50:1–10, 1995. URL `http://www.isys.uni-klu.ac.at/PDF/1995-0043-MHBM.pdf`.

[4] G. Little and R. C. Miller. Keyword programming in Java. *Automated Software Engineering*, 16(1):37–71, Oct. 2008. ISSN 0928-8910. . URL `http://link.springer.com/10.1007/s10515-008-0041-9`.

[5] A. MacCormack, J. Rusnak, and C. Baldwin. *The Impact of Component Modularity on Design Evolution: Evidence from the Software Industry*. 2007. . URL `http://www.ssrn.com/abstract=1071720`.

[6] D. Schwarz. Peeking Inside the Box: Attribute-Oriented Programming with Java 1.5, 2004. URL `http://www.onjava.com/pub/a/onjava/2004/06/30/insidebox1.html`.

[7] M. Seltzer and N. Murphy. Hierarchical File Systems are Dead. *Applied Sciences*, page 1, 2009. URL `http://portal.acm.org/citation.cfm?id=1855569`.

[8] E. Tempero, G. Bierman, J. Noble, and M. Parkinson. From Java to UpgradeJ. In *Proceedings of the 1st International Workshop on Hot Topics in Software Upgrades - HotSWUp '08*, page 1, New York, New York, USA, Oct. 2008. ACM Press. ISBN 9781605583044. . URL `http://dl.acm.org/citation.cfm?id=1490283.1490285`.

[9] M. Völter and V. Pech. Language modularity with the MPS language workbench. In *IEEE 34th International Conference on Software Engineering (ICSE)*, pages 1449–1450. IEEE, 2012. ISBN 978-1-4673-1067-3. .