The Python Programming Language as a Focal Point for Converging Research and DevOps Processes in the IDL Infrastructure

IT- and Software Engineering Tools and Methods Analysis
The Python Programming Language as a Focal Point for Converging Research and DevOps Processes in the IDL Infrastructure

AUTOR
Arne Bachmann M.A.
DLR Lufttransportsysteme
Hamburg

Deutsches Zentrum für Luft- und Raumfahrt e.V.
Institut für Lufttransportsysteme
Abteilung Integrierter Luftfahrzeugentwurf

Freigabestufe: A - Öffentlich

Hamburg, 27. Oktober 2016

Unterschriften:

Institutsdirektor: Prof. Dr.-Ing. V. Gollnick

Abteilungsleiter: Dipl.-Ing. Björn Nagel

Verfasser: Arne Bachmann M.A.
The Python Programming Language as a Focal Point for Converging Research and DevOps in the IDL Infrastructure

Arne Bachmann, Jonas Jepsen, Björn Nagel
Institute of Air Transportation Systems, German Aerospace Center
21079 Hamburg, Germany. Email: mailto:arne.bachmann@dlr.de

Abstract—While polyglot software development is a widely used approach to tackling today’s complex IT development and maintenance challenges, finding and promoting a single programming language for tasks ranging from DevOps duties to scientific analysis codes in aviation research projects bears many advantages for agile, distributed multidisciplinary design teams. This paper details advantages of employing Python as a central software tool at the Institute of Air Transport Systems and discusses its features in relation to other languages.

I. INTRODUCTION

Collaboration in complex aviation projects within multi-disciplinary teams brings together for every design or assessment task a selected number of specialists with each very different and specific demands for technical working environments, availability of software tools, computing and data resources, and organizational and computational methods. Additionally, every participant brings into meetings their personal cultural background, collaborative experience and familiarity with integrated working methodologies.

At DLR’s Institute of Air Transportation Systems (LY), inside German Aerospace Center (DLR), the existing collaborative culture of conducting complex aviation projects has been advanced into having both a versatile distributed technical infrastructure at the scientists’ disposal, and utilizing meetings regularly in focused, facilitated workshops with individuals’ access to relevant disciplinary software codes in a distributed multi-fidelity, multi-disciplinary design process at rotating sites throughout Germany, Europe and/or worldwide. To support this kind of highly communicative and interactive work technically and organizationally, integrated working spaces and interdisciplinary software integration environments were developed and set up at DLR, with the Integrated Design Laboratory (IDL) being its most prominent instantiation to support and investigate these collaboration principles at the Hamburg-Harburg site.

A. The Integrated Design Laboratory

Built to maximize meeting flexibility and collective research efficiency both on-site and when collaborating remotely, the IDL was equipped with solutions that do not necessarily maximize a single quality of the many ways to improve effective interconnected work, e.g., high-resolution visualization capabilities, specialized high-end workstations, or fast network links, but balance all aspects to find a perfect trade-off for flexible and dynamic engineering sessions, with a focus on bringing together knowledge carriers and their supporting tools and processes in a pleasant and productive atmosphere. This entailed having both cable and wireless networks available, allowing users to bring their own portable computers or workstations, providing movable working desks with integrated monitors, networking and video switches, and generally routing video signals from any source in a many-to-many fashion to the large main image wall or other target systems in and around the laboratory. The IDL thus serves systematic development and examination of improved collaboration methods for the design of air transport systems and other complex research topics [1]. Additionally, it provides the technical infrastructure for enhanced communication between engineers. Lessons learned from this kind of distributed work include intermittent co-located intensive and social sessions as described in [2]. The research results of experiments on participative multidisciplinary design optimization and the role of visualization in engineering performance in the IDL are detailed in [3], [4], [5], [6]. The IDL as a meeting and integration hub provides users with a bright high-resolution and high-contrast image wall that allows placement, routing and stretching of arbitrary digital streams and analog video signals onto it by means of hand-held tablet computers or other devices like wall terminals, cf. Figure 1. The image wall is connected to a central display driver that not only manages, decodes and feeds all video streams to the wall’s 18 cases projector modules, but may also serve as a central work station giving users access to the full display resolution to run any software they need. The versatility of a system that is fully controllable by a mobile room operator, but also able to run any custom user visualization is a key feature of the IDL’s current stage of expansion.

B. Software Integration Infrastructure

Jointly with the IDL’s meeting rooms, a technical infrastructure has been set up to provide for an on-demand computing environment for projects and meetings that use and support the IDL and plan to experiment with different forms of technical collaboration. During the construction of the lab environment, several general purpose servers selected at a Pareto optimum for maximum thread performance, number of threads and cost incurred (cf. Table I), an iSCSI Network Attached Storage (NAS) appliance with a gross capacity of 64 TB, as well as a selection of video signal converters and encoders were purchased and integrated into a unified streaming and software infrastructure, able to display video signals streamed wirelessly and digitally streams and analog video signals onto it by means of hand-held tablet computers or other devices like wall terminals, cf. Figure 1.

Fig. 1. The IDL’s high-contrast, high-resolution image wall.

1Institute website: http://www.dlr.de/ly


from network-enabled, easily rearrangeable working desks. The entire IDL concept was augmented by a network design able to switch between internal and public networks, air conditioning and heat dissipation systems, plus additional audio mixing equipment for moderated symposia, web conferences and workshops.

| Year | No. | Intel CPU model | GHz | GiB | S | C | T§ | Benchmark ||
|------|-----|-----------------|-----|-----|---|---|---|----------|
| 2011 | 3   | Xeon E7-8880v3  | 3.3-3.1 | 72   | 144 | 1628 |
| 2012 | 3   | Xeon E5-2667v2  | 3.3-4.0 | 64   | 8   | 1777 |

Table I: Evolving computing equipment of the IDL’s server room. *Year of purchase, † Number of sockets, § No. of cores, ¶ No. of threads, || CPU benchmark divided by number of cores, cf. https://passmark.com.

As the prime software integration and orchestration platform, DLR’s open-source in-house product RCE (Remote Component Environment) is mainly used, but alternative job schedulers are under investigation and may complement RCE in the future.

II. THE CHALLENGE OF AMALGAMATING RESEARCH AND IT PROCESSES

This section describes the current state of collaboration in aviation projects that meet in the IDL, and how development of the information technology (IT) infrastructure can be merged with the disciplinary design and development processes to close the gap between both worlds.

A. Engineering processes in software-centric research projects

The individual participants’ work packages and tasks within aviation projects working and meeting in the IDL had a relatively long setup phase in the past – termed the “build phase” of the engineering services, in which needed scientific codes had to be written, programs designed, and methods validated. In order to become qualified and able to perform the actual scientific computations with confidence. After that, the targeted “configuration and execution phase” can be started, which strives to answer the research questions using the tools created and are integrated with each other in the build phase. This setup process often consumed the major part of the scheduled project time frame, caused by the repeated need to debug software codes, clarify semantics of model parameters, and evolve the common design language.

Over time, however, early and in-depth communication became more recognized, and these collaborative projects [9], [11], [12] became more agile. They integrated rolling-wave planning into their project management plans, thus improving project success and become part of a single configuration management (CM) and continuous integration (CI) processes (“infrastructure as code”), all while adhering to agile practices. This is accomplished by appealing to the common goals between developers and IT operations staff alike and maximizing the use of same or similar processes and tools, thus allowing quicker response times when reacting to changes and implementing responses. As explained in [14], key ideas behind DevOps is to extend operations to development, and the other way around. In addition, it is ventured to embed one discipline into the other, again doing it in both directions.

In the case of aviation research, the researchers at DLR usually fulfill all kinds of roles including acting as developers, project managers, team leaders, and even part-time ITC managers. There are additional roles occupied by staff equipped with a different set of skills and resources, including people responsible for office and appointments management, IT planning, purchases and setup, network infrastructure development, safety and security, quality and legal concerns. Bringing together both engineers and auxiliary services is expected to result in collaboration with a deeper knowledge of the entire design process and each participant’s responsibilities, with the ultimate goal to reduce lead time, enhance quality delivered, relieve project members from peripheral and distracting duties, and lastly build up team trust and gain team maturity.

C. Choosing a universal programming language for DevOps and research applications alike

The task at hand is to find a programming language that supports both utility scripts and “serious” computing applications. Some natural candidates for this goal are the languages Python [15] and Ruby, which are at the heart of this text. Python is often termed a “scripting language”, even by the language’s designer. A purported feature of scripting languages is that they are considered more productive and suited for quick prototyping computer programs in comparison to traditional (compiled, linked) languages. A second goal considers code expressiveness, testability, and simplified concurrency, which are all features of functional programming languages and are partially supported by above two candidates, which would best be characterized as multi-paradigm languages, incorporating procedural, object-oriented (OOP) and functional (FP) programming language styles. Python includes elements of imperative, mutable state programming style, OO capabilities, list and dictionary comprehensions, iterators and generator expressions, map/filter/reduce, lambda

4 Python website: https://www.python.org
5 Ruby website: https://www.ruby-lang.org
7 Relevant quotes on OOP vs. FP: https://dZone.com/articles/what's-wrong-with-object-oriented-programming
8 https://docs.python.org/2/howto/functional.html
functions, pattern matching (limited to tuples), plus multiple return values. Where appropriate, comparison of Python and Ruby with non-scripted functional languages are referred throughout the text.

An informal survey of the programming language landscape used to create common free and open-source DevOps tools reveals the importance of Python and its main competitor in the field of dynamic “scripting” languages, Ruby; Table II provides an overview of the distribution of programming languages amongst software repositories, including the ones targeted at DevOps. Simple database queries against the source code and software hosting platforms Sourceforge and Github reveal a predominance of Python over Ruby for most keywords, with the exception of a search for the specific keyword “devops” and “configuration management” on Github. Data from the “Periodic Table of DevOps Tools” hints to a draw between the popularity of Python and Ruby, but quality of available data varies strongly and includes many missing and/or outdated links, often pointing to commercial websites of the respective tools vendors.

The difference in numbers between the former two platforms may be explained by the age and effective usage and audience of the respective sites; while GitHub has recently become popular for all kinds of collaborative software development using and integrating the distributed version control system Git. Sourceforge represents an older repository mainly used for geo-aware binary software distribution with an alleged number of 4.8 million daily downloads.

Regarding the development of language use over time, a statistical overview shows a steady increment in the number of Python-oriented repositories (with a total number of about 165,000), while the accrual of Ruby seems stagnating (133,000 projects in total). GitHub registered a total increase of 900,000 repositories between the last quarter of 2013 and 2014, but no recent data is available from that source.

For the IDL, Python was therefore selected as the best candidate for a unified DevOps and application programming language; an in-depth description of its benefits is laid out in the following sections.

D. Python introduction

Python is a versatile, open source, dynamically typed general purpose programming language with a focus on readability and syntactic brevity; this coincides with its low number of special literals like `>>>`[1], used to distinguish different language constructs, and the language’s number of reserved keywords, when compared to other languages: Erlang 27; Python (2.7) 31; C 32; Python (3.5) 33; Ruby (2.1) 41; Java 40-45; Pony 47 and C++ 62-73. A fact supporting Python’s characterization as a scripting language might be Python’s concise and compact, yet explicit writing style with cautious additions of functional elements, plus its ability to replace other scripting languages like Shell scripts easily allowing users to write operating system-level (OS) scripts in the same language as their domain code. On the other hand Python has become useful even for computational demanding problems, replacing commercial tools like Matlab[16], by utilizing prominent and popular external software libraries like “numpy”, “scipy” and “matplotlib”[17]. Although originally an interpreted language, Python has been ported to and re-implemented not only on different operating systems but also to numerous and diverse runtime environments at varying placements on a continuum from fully interpreted to compiled to machine code, including a port to the Java Virtual Machine (JVM: Python18), the .NET platform, and its Common Language Infrastructure (CLI: IronPython19), a Just-In-Time Python compiler (JIT: PyPy20), and native code compilation (py2exe), source code translation (Cython).

E. Python language features

Polyglotism in software development has been a topic discussed controversially over the past years[21] and may be seen as an essential trait of successful agile developers; on the downside, it could be perceived as unnecessary complexity that places new dependencies on projects [16], [17], and is actively discussed in the developer community[22].

<table>
<thead>
<tr>
<th>Query String</th>
<th>Top Language</th>
<th>Python</th>
<th>Ruby</th>
<th>Matlab</th>
</tr>
</thead>
<tbody>
<tr>
<td>Sourceforge</td>
<td>(total)</td>
<td>Java</td>
<td>40</td>
<td>16</td>
</tr>
<tr>
<td></td>
<td></td>
<td>C</td>
<td>74(69)</td>
<td>294</td>
</tr>
<tr>
<td></td>
<td></td>
<td>C++</td>
<td>15</td>
<td>1</td>
</tr>
<tr>
<td></td>
<td></td>
<td>cluster</td>
<td>65</td>
<td>6</td>
</tr>
<tr>
<td></td>
<td></td>
<td>devops</td>
<td>2</td>
<td>0</td>
</tr>
<tr>
<td></td>
<td></td>
<td>virtualization</td>
<td>174</td>
<td>125</td>
</tr>
<tr>
<td>GitHub</td>
<td>(total)</td>
<td>JS</td>
<td>1.1*10^6</td>
<td>1.0*10^6</td>
</tr>
<tr>
<td></td>
<td></td>
<td>C</td>
<td>125</td>
<td>0</td>
</tr>
<tr>
<td></td>
<td></td>
<td>cluster</td>
<td>3347</td>
<td>932</td>
</tr>
<tr>
<td></td>
<td></td>
<td>devops</td>
<td>Shell</td>
<td>360</td>
</tr>
<tr>
<td></td>
<td></td>
<td>virtualization</td>
<td>Shell</td>
<td>1502</td>
</tr>
<tr>
<td></td>
<td></td>
<td>Xebia Labs</td>
<td>Java</td>
<td>21</td>
</tr>
</tbody>
</table>

TABLE II

Cound of main programming languages from software hosting, develop and listing pages by query terms. Exceptional values are shown in bold font.
In the context of this paper the authors generally concur with the general propositions of developing polyglot systems. These considerations, however, might hold true mainly for developer-centric teams, which are not always at the heart of DLR’s researcher base with their large diversity of scientific backgrounds and programming experience. Therefore a consolidation of programming tools from different application areas down to a common language, being Python in this case, may help reduce training effort, sources of errors and scattering of staff skills, and serves the purpose of the DevOps concepts detailed in Section II-B for extending and embedding IT workflows with research activities.

The fields of application for Python, covered in this paper in Section III, focus on IT infrastructure maintenance, GUI development and web services for moderation and operations, and scientific user software integration into the distributed computation network using RCE. In all three areas Python helps achieving a consolidation according to the DevOps paradigm. In addition, there is a persistent popularity of Python, reliable core stability, low initial learning effort in comparison to either non-scripting languages or even Ruby, an imperative programming style with careful additions of functional-flavored program constructs, which makes Python a good candidate for DevOps convergence between IT processes and research applications.

The following paragraphs highlight certain aspects of the Python programming language and its relevance for the integration in the research infrastructure and culture:

The dynamic nature of Python programs helps users writing compact code and reduces entrance barriers. There is no need to declare variables or variable types in advance, which reduces the amount of code to write and thus may improve code readability, which is an important productivity quality, namely code cleanliness and readability:

[…] a computer language […] is a novel formal medium for expressing ideas about methodology. Thus, programs must be written for people to read, and only incidentally for machines to execute. (Abelson & Sussman) [18]

Python software distributions come with a well-documented comprehensive standard library, although it is highly fragmented, historically grown and does not follow a unified naming schema. A (typical) programming pattern is the optimization of sorting operations through the “Decorate-Sort-Undecorate” (DSU) approach. To avoid visiting elements more than once during sorting by pairwise comparisons, the decorate phase computes the sort criterion and associates it with the sequence’s objects, then sorts the sequence by these values, before restoring the original structure in the undecorate phase. It is, however, also possible to transform this pattern into just one expression. The DSU approach is useful whenever the sorting criterion of its elements is expensive to extract or compute. Python’s standard library offers both sorting variants for its sort and sorted built-in functions: a lambda function for the pairwise comparison via the cmp keyword argument, or a Schwartzian transform of the DSU pattern via the key argument that carries a function reference for the criterion extraction, which is guaranteed to be evaluated only once per element prior to sorting. Listing 1 shows the DPU pattern and a call to the optimized sort and sorted functions.

---

Listing 1. Example for the DSU pattern in Python.

Python 3 deprecated the pairwise comparison altogether, keeping only the Schwartzian transform. Ruby sorting supports DSU by its sort_by parameter.

Listing 2 demonstrates the use of list comprehensions that allow filtering or composition of list objects in a single expression without the need for explicit control structures:

---

Listing 2. List filtering and reordering operations using list comprehensions in Python. Code taken from the IDL’s server loads monitoring system.

Another example shows how to apply or compose several functions on lists of data, representing a vertical approach to Python’s map built-in function which allows an arbitrary number of sequences of function arguments:

---

Listing 3. Aggregate function application in Python.

This example can also be extended to recursive function application to chain results.

Another benefit for beginners starting to interact with Python is its selection of sensible defaults. The following listing shows the example of negative sequence (slice-) indexing, and the advantages of having memorable operator precedence that coincides with natural language use, thus avoiding most parentheses used for explicit marking of execution order:

---

Listing 4. Easy to use defaults and operator precedence in Python.

The Pony language, a recent functional programming language with a similar approach to readability and code cleanliness as Python (having, e.g., interned docstrings) has an operator precedence even easier to memorize, which is always left to right unless marked otherwise. This allows to write the following code:

---


24 Pony website: http://www.ponylang.org
After comparing Python with other languages and describing its general properties and benefits, this section demonstrates how Python as a unified tool helps in solving problems in three very distinct areas of application around the IDL, namely IT infrastructure maintenance, GUI development and web services for moderation and operations, and scientific user software integration into the distributed computation network using RCE.

III. USE CASES WITHIN THE IDL CONTEXT

The three main principles of the developing field of DevOps are communication, collaboration and integration, which match very well with the virtues of modern engineering design methodologies used in concurrent engineering (CE) or multi-disciplinary design, analysis and optimization (MDAO) processes, including the aerospace research sector that this paper is concerned with: Enhanced and intensified team and technical communication, multi- and interdisciplinary collaboration, and knowledge and software integration. The DevOps’ agile approach to unifying development and operations is therefore becoming increasingly relevant for the IDL’s ongoing and future undertakings.

Example 1: Using Python as a generic CPU loading tool. The following script utilizes a CPU at 100% by employing a busy wait performing simple operations in a loop, by means of the multiprocessing module.

```python
import multiprocessing, time

def killer():
    a = 0
    while True: a = int(float(str(a)) + 3.) / 2
    except KeyboardInterrupt: pass

def main():
    ps = []
    for x in range(2 * cpus):
        p = multiprocessing.Process(target = killer)
        p.start()
    for p in ps: p.join()

if __name__ == '__main__':
    main()
```

Lines 5–6 in Listing 6 intentionally compress the code to the utmost at the expense of cleanliness, exhibiting one limitation of the syntactical power of Python: one is not allowed to open more than one code block on any line, or have two colons on the same line of code. It is possible, however, to put a short block of code on the same line as its opening condition (line 7), although often considered a bad practice.

Figure 2 shows a screenshot of an in-house Python dashboard application for current and historic server loads display.

Example 2: In order to assess bandwidth, stability and data integrity, a tool was created that demonstrates how to concurrently write bulk data on a storage device. The data was generated by concatenating patterns of increasing byte values in blocks of 16 MiB.
as shown in listing 7: Table III shows data rate results from performance measuring with the serially attached network storage (SAN) appliance; for testing purposes 20TB of data were written to load-test the device. A similar program called "checker.py" reads back in all data and validates its contents; purpose of these tools is to trigger file system limitations or find RAID system misconfiguration, which have been reason for data loss in the past. The table shows a varying throughput depending on concurrent Python processes, highlighting effects of blocking I/O. Currently it is unclear, why data rates may exceed the theoretical bandwidth bounds of 114 MB/s\(^3^2\) when taking into account TCP and iSCSI protocol overheads, since there is no compression expected, nor caching assumed for the data streams; yet there may be an error in the implementation or assumptions. The SAN was attached to the test server by a Gigabit Ethernet network connection, however link aggregation (multipathing) was disabled on the appliance and the server not configured to support it, which would contradict these findings. Nevertheless this test confirmed a) that the storage device provides reliably storage capacity and bandwidth in the context of the IDL servers, and b) proved that Python may be used for specific kinds of I/O-bound load testing.

In order to achieve high data rates on the application level, a software and hardware combination needs to be defined and integrated that takes into account any performance-blockers in the process chain of data feeders and drains, e.g., by dumping and archiving large amounts of detailed logging data, as detailed in [19], [20], [21].

B. Meeting and moderation support tools
Example 1: Python for GUI-centric tools and moderation support. While the HPC cluster is intended to run continuously, utility software deployed on tablet computers or wall terminals around the IDL premises may be turned on and off at any time. Regardless of


<table>
<thead>
<tr>
<th>Threads*</th>
<th>Average [MB/s]</th>
<th>Total [MB/s]</th>
<th>Written (GiB)</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>209</td>
<td>209</td>
<td>37</td>
</tr>
<tr>
<td>2</td>
<td>64</td>
<td>127</td>
<td>22</td>
</tr>
<tr>
<td>4</td>
<td>38</td>
<td>153</td>
<td>27</td>
</tr>
<tr>
<td>8</td>
<td>17</td>
<td>136</td>
<td>24</td>
</tr>
</tbody>
</table>

**Table III: Write Speeds to Gigabit-Attached SAN Appliance in the IDL Server Rack, \(\ast\): Parallel Python Threads, \(\ddagger\): Average Throughput, \(\ddagger\): Total Throughput, \(\ddagger\): Total Data Written.

the fact that these portable devices are prone to sudden standby-hibernations or restarts, they are still expected to provide their services on demand whenever turned on, communicating with the image wall and streaming control systems.

Whenever an application on these devices is hibernated or put into standby-mode, its connection to the server is lost and/or its active session may be invalidated or outdated.

Therefore the detection of a wake-up from an undetermined time of standby or hibernation is needed, implemented as a regular timestamp check that may trigger a certain sequence of commands to re-establish and re-assert the previous state, or to update the application state with changes performed on the server in the meantime.

Example 2: A second facet from the application above shows how Python is able to comfortably work around faulty implementations and non-conformance with established standards in legacy systems: To communicate with the IDL’s image wall control server, a tablet application developed at LY uses the vendor’s application programming interface (API), which is provided in the form of a WSDL document\(^3^3\), which is a meta-language for web services.

Listing 9 shows part of a function description of the server’s WSDL as parsed with the SOAP\(^3^4\) library suds\(^1^5\). The entire WSDL document has a length of roughly 250.000 lines when output as a Python data structure and can therefore only be humanly digested and made sense of through informed guesses and text editor support. As can be seen from Listing 9, the service description is not self-explanatory, and while it gives a good idea

34Simple Object Access Protocol specification website, accessed Sep 1, 2016: https://www.w3.org/TR/soap12/
35Forked “suds” website: https://bitbucket.org/jurko/suds
of the data types required for each function, it lacks a general description of the function’s purpose or usage, e.g., order of calls, statefulness, guarantees, or error codes, and the function arguments’ names, requiring developers to guess and reverse-engineer required functionality.

```python
Listing 9. WSDL excerpt from the image wall control server SOAP API.
```

To provide a tangible added value and gain a real benefit over what the vendor itself provides as software tools for use in the IDL, the tablet applications need to access two network services on the target devices due to avoiding parser errors. All this is possible without the need to modify the external Python library to work around a faulty legacy system response.

Working with legacy systems, Python’s ability for dynamic code replacement makes it easy to ship code without modifying the sources it depends on, when code is imported from external modules.

"Monkey-patching" is a colloquial term for replacing a function reference by another, which opens up a way for Python to implement ideas stemming from aspect-oriented programming (AOP) [24]. Since Python supports the concept of name spaces that define a new mapping from symbols to objects whenever a module is imported into a Python program, and every imported module is an object that references its members, including all functions, it is easy to change any such reference to point to a new code object, effectively replacing the function’s implementation dynamically at runtime. This works for most places exactly like changing a variable’s reference thus effectively modifying its value, cf. Listing 11:

```python
Listing 11. Replacing module-level function references.
```

One emerging, bytecode-exclusive and/or compiled programming language to monitor in the future, equally suited for GUI development and concurrent programming, is Lever [36].

[36]Lever website: http://leverlanguage.com
C. User software integration on the HPC cluster

Example 1: Embedding scientific modeling and analysis codes into the RCE framework. For collaboration in aircraft design, LY, together with its project partners, has developed the open data exchange format “Common Parametric Aircraft Configuration Schema” (CPACS)\(^{37}\), which serves as the central technical language for information exchange between disciplinary software tools in multi-disciplinary aviation collaboration projects, cf. [11]. The RCE framework not only allows defining software wrappers for arbitrary programs, but also provides special support for the extraction and manipulation of single design parameters from hierarchical, XML-based CPACS datasets. Nevertheless, many of these integrated software used in the analysis workflows have been developed at DLR over a long time and were made CPACS-compatible only recently. Here Python simplifies the wrapping process, as it allows writing simple wrappers around existing code in order to extract specific configuration items and switches that steer the software’s execution. Therefore Python is part of almost any scientific code present in the RCE network, even if the software itself is written in another programming language or has been compiled into a binary executable.

Example 2: Virtual environments for Python. When developing several projects at the same time, keeping track of Python package dependencies can be difficult. When working with a single Python installation for the development of multiple applications it can be difficult to keep track of all the python packages which are used by each of them. It is even more difficult when there are conflicts in the dependencies such that for application “A” a specific version of a Python package is required and for application “B” a different one. Under these circumstances Python’s support for virtual environments is invaluable and can be seen as a simple containerization solution, in contrast to operating system level or machine level virtualizations.

By making use of virtual environments each application can run in its own Python environment containing a minimum of packages required for the application to run. At LY the conda\(^{38}\) package management system with its integrated support for virtual environments is used. On the example of the open source conceptual aircraft design software VAMPzero\(^{39}\) the use of virtual environments for developing and testing software is presented. The development of VAMPzero is done in its own virtual environment. A new virtual environment is created by:

```bash
conda create -n VZ python=2.7 numpy matplotlib scipy networkx lxml sphinx
```

Listing 13. Command-line example to create a virtual Python development environment.

This creates a new Python 2.7 environment with the name VAMPzero_dev. The list of Python packages given with the command are installed into the environment which make it immediately ready for use.

The environment can either be activated for use in a shell or configured in an integrated development environment (IDE) such as PyCharm\(^{40}\). Even more useful than for development is the use of a virtual environment for testing of releases. When a version is ready for release, e.g. as a Python wheel file\(^{41}\), the generated wheel can thus be tested in a clean environment. Dependencies given in the packages setup.py would automatically be installed. After a successful installation procedure, other test cases can be run on the clean system as a validation for the wheel correctness.

Example 3: Generic code wrapping. The survey from Section II-C already showed a significant predominance of Python over Ruby for DevOps tools. Two further analyses from Google Trends\(^{42}\), cf. Figure 4, and the often cited TIOBE Index of programming languages\(^{43}\) underpin this observation: In the August 2016 index, Python scored fifth place with 4.4% appearance and Ruby twelfth with 2.3%. This might hint to an easier adoption for new Python users. By means of its highly popular third-party libraries numpy, scipy, matplotlib, Python might serve almost as a drop-in replacement for the Matlab language with similar ease of matrix and list operations and visualization capabilities, but freely available.

Concurrently with the construction of the IDL the processes for integrating and managing user code into the distributed computing platform have been developed, tested, and revised repeatedly.

Example 2: Embedded Python: The RCE platform for integration, automation, collaboration and data management comes equipped with software wrappers for the Python language, delivered in two flavors: Binary Python installations are supported, and require only a manual setup for Python’s interpreter path for the first time each workflow file is opened. Interestingly though, due to the fact RCE is build on Java technology, the language’s JVM implementation “Jython” is shipped directly with RCE, allowing a zero effort script integration into user workflows. The drawback found in this context, however, is the fact that RCE currently ships only with a highly outdated Jython version equivalent to Python’s standard library version 2.5.1, which lacks many (backported) library additions present in the 2.7 and 3.x versions.

D. Writing functional-flavored Python code

For the sake of readability and conciseness, it may be beneficial to agree on breaking some conventions communicated in the Python community. The underscore character _ for example, in Python marks methods as implicitly private (not exported), and on the interactive Python shell always carries the last computed expression’s return value. Due to its unique appearance in source code, however, the underscore may be used for other purposes quite elegantly:

Example 1: When attempting to move from an imperative, destructive, hard to test and to parallelize to a functional and collection-centric programming style, the underscore may serve as a placeholder representing the current element inside a list comprehension:

```python
myServer = [x for x in servers if x.name == 'node1'][0]
```

Listing 14. Example code for a possible use of underscore in list comprehensions, taken from a HPC configuration management tool used on the HPC cluster.

37CPACS website: http://cpacs.de

38Conda introduction: http://conda.pydata.org/docs/intro.html

39VAMPzero website: https://software.dlr.de/p/vampzero/

40PyCharm website, accessed Sep 1, 2016: https://www.jetbrains.com/pycharm/

41Python Wheels website: http://pythonwheels.com

42Google Trends website: https://www.google.de/trends/

43TIOBE Index website: http://www.tiobe.com/tiobe-index/
Compare this with an Erlang code example, which has a more math-inspired syntax and uses the underscore only as a throw-away placeholder in patterns, cf. also Listing 19 (here Server# marks variable S as a specific record type):

```
hd([S | S = servers, Server#.name = hostname]).
```

Listing 15. Example code for list comprehensions in Erlang.

In other cases, the use of underscore might not be appropriate, e.g., when naming the intermediate variables by their generic meaning like “key” and “value”:

```
{key, val} = [k, v in d.items()] |
```

Listing 16. Example code for a possible naming inside list comprehensions while iterating over a dictionary.

Example 2: When writing OO programs in Python, it may feel cumbersome to declare the (almost implicit) “self”-reference of the current object in the signature of every method that object exposes. Traditionally called “self”, the first function parameter is usually understood by IDEs through support of auto-completion and text suggestions; the Python object model always automatically and silently prepends the current object’s self-reference to any method call of that object (via dot notation). Making it a habit to use the underscore instead, as used in Listing 8, however, may improve method declaration and readability instantly, as there is a visual discriminability between named variables and the (unnamed) self-reference, invoke other methods (as in _.getName()) and refer to objects’ attributes (as in _.name).

A combination of above two suggestions for code clarity, however, will lead to hard to track errors; therefore the use of a double-underscore for the list comprehension may be considered as in the following code:

```
\[
\text{safeValue} = \text{getValue}(x) \text{ if } x \text{ is not None holds true (is being the identity comparison operator), otherwise assign it the (exceptional) value of 0. This provides a clear hint about what is the norm or expected program flow, but also defines a standard alternative (fallback) after it, making it easy to understand by humans reading the code, as it emphasizes the important (or “unmarked”) default case before mentioning the (less likely) exceptional case. Interestingly, most languages implement a different order of arguments to the ternary expression, with one other exception beside Python being Fortran, cf. Table IV: No matter how it is implemented syntactically in any language, most developers agree in that nesting ternary operators is a bad practice to avoid.}
```


```
convention = \{ \text{for } k, v \text{ in } d.items()} \}
```

Listing 17. List and dictionary comprehensions using underscores in private, intermediate run variables.

Example 3: **Conditional (or ternary) expressions** (or operators) are sometimes frowned upon for obscuring code readability, but match the expressive programming style known from functional programming languages when trying to achieve something similar in Python44.

```
safeValue = getValue(x) if x is not None holds true (is being the identity comparison operator), otherwise assign it the (exceptional) value of 0. This provides a clear hint about what is the norm or expected program flow, but also defines a standard alternative (fallback) after it, making it easy to understand by humans reading the code, as it emphasizes the important (or “unmarked”) default case before mentioning the (less likely) exceptional case. Interestingly, most languages implement a different order of arguments to the ternary expression, with one other exception beside Python being Fortran, cf. Table IV: No matter how it is implemented syntactically in any language, most developers agree in that nesting ternary operators is a bad practice to avoid.
```

of reduced training effort, higher code quality, easier debugging and setup of development environment. Python as a single unified tool for these diverse jobs provides users both from the converging DevOps and research staff with a powerful, versatile and easy to learn programming language. Administration and maintenance of Python distributions is easy, and almost any operating system is supported, while different distributions are optimized for aspects ranging from performance over interoperability to concurrency guarantees.

Ruby came about some years later than Python, with a different design philosophy, but was initially at a disadvantage with low performance and very much linked with its landmark convention-over-configuration web framework Rails. These early drawbacks have mostly been resolved [25] and there are even JIT implementations of Ruby under way [60], however in the context of the needs and historic education of most the IDL’s staff Python is deemed a much better candidate selected for all the tasks and use cases documented above.

A. Outlook

The transition to infrastructure-as-code in the IDL is still ongoing, and collaboration processes are not yet mature enough for every participant to allow for a full on-demand IT self-service. These goals, however, have been laid down and are part of the laboratory and working methodologies evolution, both a central part of LY’s current and future development.

To name a mundane software engineering decision and educational challenge, the migration of existing, and a best practice recommendation for future code to the new language variant Python 3 will become a major activity in the future. Although largely source compatible, the language overhaul entails several syntactic changes that require sometimes restructuring or reimplementation of certain code constructs. Due to extensive library and platform support most developers still write their code in Python 2.x at LY. Whoever wants to enjoy the benefits of the new language and its library additions, however, is recommended to migrate to Python 3, as only certain security fixes and features are backported to the 2.7 branch.

Further consideration on improvements around the IDL infrastructure and user processes has to go into ubiquitous monitoring and logging to gather data for better and more agile decision making, observing security and privacy concerns, invest into more reproducible build automation and release management, provide more on-demand services provisioning (platform-as-a-service), and provide integrated training on the IT and research convergence.

REFERENCES


