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

DLR

ILT
Institut für Lufttransportsysteme

Institutsbericht
IB-328-2016-25

TITEL
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

12 Seiten

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 multi-disciplinary 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)[1] 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 cased 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.

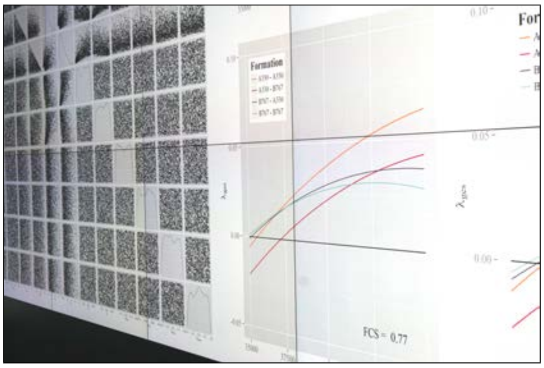

Fig. 1. The IDL's high-contrast, high-resolution image wall.

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[2] 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

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

[2]iSCSI: Internet Small Computer Systems Interface. https://en.wikipedia.org/wiki/ISCSI

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§	Bechmark‖
2011	1	Core i7-2600	3.4 - 3.8	16	1	4	8	1921
2012	3	Xeon E5-2667v2	3.3 - 4.0	64	2	8	16	1777
2015	1	Xeon E7-8880v3	2.3 - 3.1	256	4	72	144	1628

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)[3] [7], [8] 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 [9] –, in which needed scientific codes had to be written, programs designed, and methods validated [10], 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 stakeholder engagement by means of regularly scheduled technical assessments, facilitated workshops, social events, and a generally layered approach to improving tool design fidelity and reliability throughout the project schedule. Aside from the individual projects' contents, DLR's infrastructure for software integration using RCE was further improved and published as open-source to gain more overall visibility and encourage project partners to contribute.

For effective meetings that often cover many topics in a short amount of time, a reliable network and computing environment was especially important for the IDL, to serve both users and the network of integrated scientific analysis codes. Those computing resources enable users and meeting organizers to prepare design studies either on the eves of an upcoming meeting or overnight between the scheduled meeting days. In addition, participants are enabled to connect to their agency's offices' computer desktops to check on their codes or data repositories when needed, while at the same time being part of a larger distributed workload computation process.

[3] RCE website: http://rcenvironment.de

B. DevOps in the IDL

To improve the user experience for these meetings and reduce lead time for the introduction of new tools, project's user groups, and computational studies, the IDL staff decided to slowly introduce and commoditize process principles that can be subsumed under the term of "DevOps" [13].

In the DevOps movement IT operations and development converge 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 and Ruby[4], which are at the heart of this text. Python is often termed a "scripting language", even by the language's designer[5]. 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 [15]. 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[6]. Python includes elements of imperative, mutable state programming style, OO capabilities, list and dictionary comprehensions, iterators and generator expressions[7], map/filter/reduce, lambda

[4] Python website: https://www.python.org
Ruby website: https://www.ruby-lang.org
[5] Wikipedia article on Python, accessed Aug 25, 2016:
https://en.wikipedia.org/wiki/Python_(programming_language)
[6] relevant quotes on OOP vs. FP:
https://dzone.com/articles/whats-wrong-with-object-oriented-programming
[7] 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[8] and Github[9] 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[10]. Data from the "Periodic Table of DevOps Tools"[11] 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[12], 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[13] 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 quarters 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.

Query String	Top Language	Python	Ruby	Matlab
Sourceforge				
(total)	Java (53884)	16741	2076	1499
CM	Java (40)	**16**	5	0
science	Java (9349)	3474	263	818
scientific	C++ (7469)	2947	194	750
hpc	C (23)	15	1	0
cluster	C (135)	65	6	7
devops	Python	2	0	0
virtuali[z/s]ation	Java (174)	125	22	0
GitHub				
(total)	JS (2.3 mio)	$1.1*10^6$	$1.0*10^6$	56.772
CM	Ruby	96	**125**	0
science	R (5870)	3775	63	29
scientific	Python	730	23	31
hpc	C (374)	235	0	20
cluster	Python	3347	932	237
devops	Shell (652)	360	**535**	0
virtuali[z/s]ation	JS (2276)	1502	861	22
containerization	Shell (150)	45	23	0
Xebia Labs	Java (21)	**14**	13	0

TABLE II
COUNT OF MAIN PROGRAMMING LANGUAGES FROM SOFTWARE HOSTING, DEVELOP AND LISTING PAGES BY QUERY TERMS. EXCEPTIONAL VALUES ARE SHOWN IN BOLD FONT.

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 `:;,%[]` used to distinguish different language constructs, and the language's number of reserved keywords, when compared to other languages: Erlang 27[14]; Python (2.7) 31; C 32; Python (3.5) 33; Ruby (2.1) 41; Java 40-45; Pony 47 and C++ 62-73[15]. 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 many 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: Jython[18]), the .NET platform and its Common Language Infrastructure (CLI: IronPython[19]), a Just-In-Time Python compiler (JIT: PyPy[20]) 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].

[8] Sourceforge website, Aug 23, 2016: https://sourceforge.net
[9] GitHub website, accessed Aug 23, 2016: https://github.com
[10] GitHub website query example, accessed Aug 24, 2016: https://github.com/search?q=devops
Results may differ by day of query and depend on exact query phrasing, e.g., language mentiond vs. file extension, using the `language:` prefix
[11] Xebia Labs' Periodic table of DevOps tools, accessed Sep 5, 2016: https://xebialabs.com/periodic-table-of-devops-tools/
Survey procedure: For each category in the tools matrix, every list of tools was checked. For every tool with a working Wikipedia link the main programming language was looked up. Tools with professionally written Wikipedia entries that listed mainly company information were disregarded. Counting includes all mentions of a language, removing duplicate tool listings for different categories.
[12] Git website: https://git-scm.com
[13] GitHut website, accessed Sep 1, 2016: http://githut.info
[14] http://erlang.org/doc/reference_manual/introduction.html
http://tutorial.ponylang.org/appendices/keywords.html
http://s.dlr.de/hb3c ; http://s.dlr.de/w38x ; http://s.dlr.de/tmf3
http://en.cppreference.com/w/cpp/keyword
[15] `import keyword; print(len(keyword.kwlist))`
[16] Matlab website: http://www.mathworks.de/products/matlab/
[17] NumPy, SciPy, and Matplotlib websites: http://www.numpy.org, https://www.scipy.org, http://matplotlib.org
[18] Jython website: http://www.jython.org
[19] IronPython website: http://ironpython.net
[20] PyPy, Py2Exe, Cython websites: http://pypy.org, http://www.py2exe.org, http://cython.org
[21] Blog of Neal Ford from Dec 05, 2006, accessed Aug 25, 2016: https://memeagora.blogspot.de/2006/12/polyglot-programming.html
– Article by Andres Almiray on Dzone website from Jun 04, 2008: https://dzone.com/articles/how-many-times-are-we-going-ki
[22] Conference website, accessed Aug 29, 2016: http://polyconf.com

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[23] 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.

```
>>> decorated = [(student.grade, i, student) for i, student in enumerate(
  ↪ students)]
>>> decorated.sort()
>>> [student for grade, i, student in decorated]          #
  ↪ undecorate
[('john', 'A', 15), ('jane', 'B', 12), ('dave', 'B', 10)]
```

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 Python lists in a single expression without the need for explicit control structures:

```
loads = \
  + [(0, load, "System") for pid, load, name in P.values() if pid == 0]
    [(pid, load, name)   for pid, load, name in P.values() if pid != 0]
```

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:

```
def applyOrNone(func, *args, **kwargs):
  '''
  Apply a function and catch errors into None return.
  >>> print applyOrNone(lambda a,b: 1/0, 1, 2)
  None
  >>> print applyOrNone(lambda c = 0: c + 1, c = 5)
  6
  '''
  try: return func(*args, **kwargs)
  except Exception, e: import sys; print >> sys.stderr, "Wrapped␣exception␣'%s'"
    ↪ % e; return None

def applyAll(funcs, argss = [], kwargss = []):
  '''
  Apply different values to a collection of functions.
  >>> applyAll([lambda a: 1/a, lambda a: 2*a],\
               [(0,), (2,)])
  [None, 4]
  >>> applyAll([lambda a: 1/a, lambda a: 2*a],\
                kwargss = [{'a':0}, {'a':2}])
  [None, 4]
  '''
  return [applyOrNone(func, *args, **kwargs)
    for func, args, kwargs in zip(
        funcs,\
        argss if len(argss) == len(funcs) else [()] * len(funcs),\
        kwargss if len(kwargss) == len(funcs) else [{}] * len(funcs)
    )
  ]
```

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:

```
>>> a, b, c = [1, 2], [1], []
>>> print a[:-1], b[:-1], c[:-1] # slicing for "all but the last element"
[1], [], []
>>> inbounds = lambda x0, y0, x, y, w, h: x0 >= x and y0 >= y and x0 < x + w and
  ↪ y0 < y + h
>>> print inbounds(20, 0, 0, 0, 100, 100), inbounds(0, 100, 0, 0, 100, 100)
True False
```

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

The Pony language[24], 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:

[23]Archived original post, accessed Sep 3, 2016: http://www.stonehenge.com/merlyn/UnixReview/col64.html

[24]Pony website: http://www.ponylang.org

```
a = b = a    # works in Pony
a, b = b, a  # works in Python
```

Listing 5. Code example for swapping variables in Pony and Python.

Observing Ruby code, it optionally allows to leave out parentheses around function arguments; additionally it does not have Python's block opening marker **:** nor syntactical indentation, thus requiring an explicit block-terminating keyword `end`.

Regarding **convenience and interoperability**, Python's standard library comes with a vast number of modules ranging from very specific OS level manipulation as in `posix`, `socket`, file archive support and compression algorithms as in `bz2`, `gzip`, `zipfile`, network protocol support as in `ftplib`, `telnetlib`, `xmlrpclib`, to Python-centered modules as in `abc`, `ast`, `dis`.

Although there is no directly parsable **structured markup** as JavaScript has with JSON[25], Python supports parsing JSON by means of the standard module `json`. XML data can be parsed by several implementations of streaming and DOM[26] parsers `xml.dom`, `xml.sax`; at LY the external library `lxml`[27] is used a lot, which provides an intuitive OO access to XML documents. For parsing YAML structures, which are used in many DevOps tools like Ansible[28], Chef[29] or Puppet[30], however, an external library has to be installed, while Ruby supports it by default.

For **unit tests and source documentation**, Python comes with two very useful features: Unlike Java, where documentation from the source code isn't persisted in its `.class` bytecode files, in Python every block of code may be augmented by a simple string that contains information about it, which can either be extracted as browsable documentation or be queried during runtime by a simple attribute access. Secondly, the standard library contains the `doctest` module, which allows to place unit test code into these docstrings. This serves as usage specification, and puts the test code very close to the tested code, which has advantages especially when shipping small units of self-sufficient code.

In conclusion, the Python programming language lends itself to support both researchers and DevOps endeavors quite naturally, as it exhibits explicit encoding of behavior in an imperative programming style without too much implicit language "magic"[31], e.g., avoiding larger amounts of "convention over configuration" that require programmers to memorize and eventually understand many underlying assumptions and mechanics of the software system to benefit from the actual convenience these frameworks promise. While the Ruby language offers a more functional programming style and is suitable for utilization as a domain specific language (DSL) for many configuration needs, Python is considered easier to adopt by students that have learned an imperative (and OO) first programming language, and by engineers familiar with various languages and environments including the matrix-oriented programming style encountered in the Matlab environment and language, which might as well largely be replaced by Python and its accompanying libraries.

III. Use cases within the IDL context

After comparing Python with other languages and describing its general properties and benefits, this section demonstrates how

[25]JSON website: http://www.json.org
[26]Document Object Model: http://s.dlr.de/8lqg
[27]lxml website: http://lxml.de
[28]Ansible website, accessed Aug 25, 2016: https://docs.ansible.com
[29]Chef website, accessed Aug 25, 2016: https://www.chef.io
[30]Puppet website, accessed Aug 25, 2016: https://puppet.com
[31]Chris Oliver, Ruby vs. Python, accessed Sep 1, 2016: http://learn.onemonth.com/ruby-vs-python

Fig. 2. Screenshot of the server loads dashboard application.

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.

A. HPC cluster monitoring and administration

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 uniting 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.

```
1  import multiprocessing,time
2
3  def killer():
4    a = 0
5    try:
6      while True: a = int(float(str(a)) + 3.) / 2
7    except KeyboardInterrupt: pass
8
9  if __name__ == '__main__':
10   cpus = multiprocessing.cpu_count()
11   ps = []
12   for cpu in range(cpus):
13     P = multiprocessing.Process(target = killer)
14     ps.append(P); P.start()
15   try:
16     while True: time.sleep(.1)
17   except KeyboardInterrupt: pass
```

Listing 6. CPU-loading script "killer.py".

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,

```
import multiprocessing,os,signal,sys,time,threading,uuid

maxProcesses = 4
runFor = 60 * 60 # 1 hour in seconds
kibi = 1024L
mebi = kibi ** 2
gibi = kibi ** 3

def filler(index, data): # concurrent writer
  assert len(data) == 16 * mebi
  st = long(time.time()); count = 0L # every increment equals len(data) bytes
  try:
    with open(uuid.uuid4().hex + ".dat", "wb") as fd:
      while True: fd.write(data); count += 1L
  except KeyboardInterrupt: time.sleep(index * 1.4) # linearize output after
    ↪ break
  except Exception: print "Error"
  finally: print "Process", index, "wrote", count * long(len(data)) / gibi, "GB 
    ↪ at", count * long(len(data)) / (mebi * (long(time.time()) - st)), "MB/s"

if __name__ == '__main__':
  st = time.time()
  cpus = multiprocessing.cpu_count(); print "%3d cores found" % cpus
  cpus = min(maxProcesses, cpus); print "Using %3d cores" % cpus
  data = ("".join([chr(i) for i in range(256)])) * 256 * 256 # 16MB of
    ↪ consecutive 256 different bytes)
  ps = [] # TODO use multiprocessing.Pool instead
  for cpu in range(cpus):
    P = multiprocessing.Process(target = filler, args = [cpu, data])
    ps.append(P); P.start()
  try:
    while True:
      time.sleep(.1)
      if (time.time() - st) > runFor:
        threading.Thread(target = os.kill, args = [os.getpid(), signal.
          ↪ CTRL_C_EVENT if sys.platform == 'win32' else signal.SIGINT]).start
          ↪ ()
        time.sleep(maxProcesses / 2)
        break
  except KeyboardInterrupt: pass
  finally: print "Ran for", int(time.time() - st) / 60, "minutes"
```

Listing 7. Storage-loading script “filler.py”.

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 20 TB 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 throughout 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[32] 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 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.

[32]Serverfault FAQ website: https://serverfault.com/questions/301505/iscsi-transfer-rate-using-standard-gigabit-networking

Threads*	Average (MiB/s)†	Total (MiB/s)‡	Written (GiB)§
1	209	209	37
2	64	127	22
4	38	153	27
8	17	136	24

TABLE III
WRITE SPEEDS TO GIGABIT-ATTACHED SAN APPLIANCE IN THE IDL SERVER RACK. *: PARALLEL PYTHON THREADS, †: AVERAGE THREAD THROUGHPUT, ‡: TOTAL THROUGHPUT, §: TOTAL DATA WRITTEN.

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.

```
class AtIntervals(threading.Thread):
  def __init__(_, callback, interval = 0.7):
    _.callback = callback
    _.interval = interval
    _.stopped = threading.Event()
    threading.Thread.__init__(_)

  def run(_):
    while not _.stopped.wait(_.interval): _.callback()

  def stop(_):
    _.stopped.set()


class StandbyDetector(object):
  def __init__(_, callback, interval = 1.5):
    _.periodic = AtIntervals(_._checkWakeUp, interval)
    _.callback = callback
    _.interval = interval
    _.lastCheck = time.time()
    _.periodic.start()

  def _checkWakeUp(_):
    newTime = time.time()
    if (newTime - _.lastCheck) > _.interval * 1.1: # give ten percent
      _.callback() # notify app about wakeup
      _.lastCheck = time.time() # new timestamp because callback may run for a
        ↪ longer time
    else:
      _.lastCheck = newTime

  def stop(_):
    _.periodic.stop()
```

Listing 8. Wake-up recognition for GUI updates from recent server state.

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[33], 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[34] library `suds`[35].

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

[33]Web Services Description Language website 1.1 from Mar 15, 2001, accessed Aug 25, 2016: https://www.w3.org/TR/wsdl

[34]Simple Object Access Protocol specification website, accessed Sep 1, 2016: https://www.w3.org/TR/soap12/

[35]Forked “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.

```
createPerspective2 =
   (Operation){
      name = "createPerspective2"
      tns[] =
         "tns",
         "http://openapi.cms.barco.com/",
      input =
         (Message){
            name = "createPerspective2"
            qname = "(createPerspective2,␣http://openapi.cms.barco.com/)"
            parts[] =
               (Part){
                  root = <wsdl:part type="tns:sessionId" name="arg0"></wsdl:part
                    ↪ >
                  name = "arg0"
                  qname[] =
                     "arg0",
                     "http://openapi.cms.barco.com/",
                  element = "None"
                  type = "(u'sessionId',␣u'http://openapi.cms.barco.com/')"
               },
```

⋮

```
         }
      output =
         (Message){
            name = "createPerspective2Response"
            qname = "(createPerspective2Response,␣http://openapi.cms.barco.com/
              ↪ "
            parts[] =
               (Part){
                  root = <wsdl:part type="tns:objectId" name="return"></wsdl:
                    ↪ part>
                  name = "return"
                  qname[] =
                     "return",
                     "http://openapi.cms.barco.com/",
                  element = "None"
                  type = "(u'objectId',␣u'http://openapi.cms.barco.com/')"
               },
         }
      faults[] = <empty>
   }
```

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 different systems through SOAP and REST (Representational State Transfer) APIs respectively. By combining these two services, the IDL software development allows users to make maximum use of the state-of-the-art hardware systems themselves, which only lack problem-oriented software solutions. Figure 3 gives an impression of the highly simplified user interface that enables meeting moderators to freely route and place arbitrary wired or wireless image streams from IDL rooms on the central image wall, thus replacing and surpassing the vendor's unwieldy standard software.

Python serves here as a means for evolving and augmenting legacy software and hardware systems, which cannot be modified by its users or whose creator does not take interest in opening its tools; cf. [22] as an example for problems arising from software that had not been designed with steady evolution in mind and which is oblivious to its context of application. The specific workaround found to parse a faulty server WSDL was a) to use Python's ability to (monkey-)patch a method in the `ssl` sub-module of a strictly validating implementation of the third-party XML parser suds, b) to replace all break tags `<BR>` by the standards-conform form `<br />` (cf. below), and c) to remove excess characters after the last closing tag to avoid parser errors. All this is possible without the need to modify and distribute additional source code on the target devices due to Python's module system and dynamic nature, plus treating functions as first-class objects (also termed "first-class citizens" [23]).

Fig. 3. Screenshots of the simplified (left) and comprehensive (right) video signal routing app.

```
import suds
import suds.reader
# Because Barco CMS delivers non-wellformed XML with exceeding characters that
  ↪ break parsing, we need to patch SUDS's SAX parser:
def monkeyPatchedSudsDownload(_, url):
  content = None  # this is original code
  store = _.options.documentStore
  if store is not None: content = store.open(url)
  if content is None:
    fp = _.options.transport.open(suds.transport.Request(url))
    try: content = fp.read()
    finally: fp.close()
  ctx = _.plugins.document.loaded(url=url, document=content)
  content = ctx.document
  # Now fix buffer bug: this is new code
  last = -1
  while content[last] != '>': last -= 1
  if last < -1: content = content[:last + 1]
  # Now fix lazy tags
  import re; regex = re.compile('([a-zA-Z]+)=([a-zA-Z0123456789]+)([>␣\\n])')
  content = regex.sub('\\1="\\2"\\3', content).replace('<BR>', '<BR/>')
  sax = suds.sax.parser.Parser()
  return sax.parse(string = content)
suds.reader.DocumentReader.download = monkeyPatchedSudsDownload # patch the code
import suds.client  # now load the module that uses the patched method
```

Listing 10. Definition of a code replacement for parts of an 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:

```
>>> import math
>>> print(round(math.sin(1), 1))
0.8
>>> math.sin, math.cos = math.cos, math.sin # do not do this!
>>> print(round(math.sin(1), 1))
0.5
```

Listing 11. Replacing module-level function references.

The REST operations used to contact the wireless video transmission appliance are easily implemented in Python:

```
import json, urllib2
j = json.loads(urllib2.urlopen(url).read()) # GET

request = urllib2.Request(url, data = 'value=%s' % value)
request.get_method = lambda: 'PUT'
j = json.loads(urllib2.urlopen(request).read()) # PUT
```

Listing 12. REST-ful operations GET and PUT with the Python standard library, without error handling.

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:

```
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.

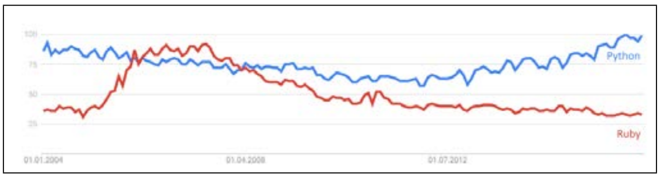

Fig. 4. Screenshot from a trends comparison query for Python and Ruby, restricted to the category “Computer and Technology”.

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 may 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:

```
# Extracts server with a matching name (assuming there is one)
myServer = [_ for _ in servers if _.name == nodeName][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.

[37] CPACS website: http://cpacs.de
[38] Conda introduction: http://conda.pydata.org/docs/intro.html
[39] VAMPzero website: https://software.dlr.de/p/vampzero/
[40] PyCharm website, accessed Sep 1, 2016: https://www.jetbrains.com/pycharm/
[41] Python Wheels website: http://pythonwheels.com
[42] Google Trends website: https://www.google.de/trends/
[43] TIOBE Index website: http://www.tiobe.com/tiobe-index/

Compare this with an Erlang code example, which has a more maths-inspired syntax for the same task 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#S.name == NodeName]).
```

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":

```
converted = [(int(k), str(v)) for 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:

```
asInts = [int(__) for __ in aList]
asReverse = {_v: int(_k) for _k, _v in maps.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 Python[44]:

```
safeValue = getValue(x) if x is not None else 0
protected = lambda A, B: A / B if B != 0. else 0. # as in Pony language
```

Listing 18. Ternary/Conditional expression/operator in Python.

This may be read as "assign to the variable `safeValue` by default the result of the expression `getValue(x)` if the condition `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.

Language	Conditional Expression Order						
C, Java, Haskell, ...		COND	?	THEN	:	ELSE	;
Pony	if	COND	then	THEN	else	ELSE	end
Python		THEN	if	COND	else	ELSE	
Fortran	merge(	THEN	,	ELSE	,	COND	)

TABLE IV
CONDITIONAL EXPRESSION ELEMENT ORDER FOR DIFFERENT PROGRAMMING LANGUAGES. COND REPRESENTS THE CONDITION EXPRESSION, THEN AND ELSE THE EXPRESSIONS OR STATEMENTS THAT ARE CONDITIONALLY EVALUATED.

In Erlang, as in most functional programming languages , there exists no special ternary expression, but every block of code is an expression in its own right. The `if` (and `case`) expression allows to evaluate conditions which might act as a ternary (or higher count) conditions, while the compiler ensures that the different if clauses always guarantee a result or raise an exception. An example for such a function body is given in the following Erlang code:

```
saxHandler({startElement, _, "link", _, Attrs}, _, {Entries, inItem, Entry}) ->
  case lists:any(fun(Elem) -> element(3, Elem) == "href" end, Attrs) of  %
    ↪ sometimes href contains link
    true ->
      {Entries, inItem, Entry#entry{url = case Entry#entry.url of undefined ->
        ↪ helpers:removeUtmSource(element(4, hd(lists:filter(fun(Elem) ->
        ↪ element(3, Elem) == "href" end, Attrs)))); Any -> Any end}};
    false ->
      {Entries, inItemLink, Entry}  % go into sax loop and wait for character
        ↪ content
  end;
```

Listing 19. Case expression in Erlang, example from a news feed parser.

Pony implements a C-style conditional with a block structure as in Erlang[45]. Python again takes an interesting hybrid approach by allowing even left-side conditional expressions, e.g., for making conditional behavioral selection, as follows:

```
# Example 1: Conditionally calling one of two functions
result = (sophisticatedFunc if supported else simplifiedFunc)(funcArgs)

# Example 2: Calling an indexed function from a dict
result = myDict[myKey](functionArguments) # similar use of indirection
```

Listing 20. Use of left-side conditional expressions in Python.

Example 4: **Anonymous inline functions**, marked by the `lambda` keyword, helps in writing collection-centric and functional-style programs in Python, since they have both a dense syntax by leaving out the return statement, can be defined anonymously at any place in the code. Lambda expressions provide a similar expressiveness as normal function declarations using the `def` keyword and just as named functions allow for positional and keyword arguments plus default values, but accept only one expression in their immediate function body, disallowing multiple expressions or statements chained by semicolons. Side effects are, however, not prohibited and left to the developer's discretion how and if to avoid them. Listing 21 shows uses of lambda uses:

```
> filter(lambda elem: elem.hasProperty(), listOfElements) # by predicate
> map(lambda elem: int(elem), listOfStrNumbers) # apply to each element
> [int(elem) for elem in listOfStrNumbers] # equivalent, but faster
> sorted(myList, lambda a, b: cmp(a[1], b[1])) # sorts by sub-element
```

Listing 21. Use of left-side conditional expressions in Python.

IV. CONCLUSION

Avoiding polyglotism and focusing on one programming language for a certain spectrum of applications from IT operations and infrastructure software development on one side, aviation research and efficient workshop support on the other side, yields the advantages

[44]Original poll from Guido van Rossum, accessed Aug 25, 2016: http://legacy.python.org/dev/peps/pep-0308/

[45]Tutorial, accessed Sep 1, 2016: http://tutorial.ponylang.org/expressions/control-structures.html

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[46], 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

[1] A. Bachmann, J. Lakemeier, and E. Moerland, "An integrated laboratory for collaborative design in the air transportation system," in 19th ISPE International Conference on Concurrent Engineering, Trier, Germany, 2012, pp. 1–2. [Online]. Available: http://www.ce2012.org

[2] B. Nagel et al., "Virtual aircraft multidisciplinary analysis and design processes – lessons learned from the collaborative design project vamp," in 4th CEAS Air & Space Conference, Flygtekniska Förening, Linköping, Sweden, 2013.

[3] E. Dineva et al., "Empirical performance evaluation in collaborative aircraft design tasks," in 20th ISPE International Conference on Concurrent Engineering, 2013, pp. 110–118. [Online]. Available: http://ebooks.iospress.nl/book/20th-ispe-international-conference-on-concurrent-engineering

[4] ——, "New methodology to explore the role of visualisation in aircraft design tasks: An empirical study," International Journal of Agile Systems and Management (IJASM), vol. 7, no. 3/4, pp. 220–241, 2014, DOI: 10.1504/IJASM.2014.065356. ISBN online: 1741-9182; print: 1741-9174. ISSN 1741-9174. [Online]. Available: http://www.inderscience.com/info/inarticle.php?artid=65356

[46]Rubinius website, accessed Aug 30, 2016: http://rubinius.com
JRuby website, accessed Aug 30, 2016: http://jruby.org
Topaz website, accessed Aug 30, 2016: http://www.topazruby.com
RuJIT GitHub website, accessed Aug 30, 2016: https://github.com/imasahiro/rujit
InfoQ webcast, accessed Aug 30, 2016: https://www.infoq.com/presentations/graal-trufle

[5] ——, "Human expertise as the critical challenge in participative multidisciplinary design optimization – an empirical approach," in Moving Integrated Product Development to Service Clouds in the Global Economy Advances in Transdisciplinary Engineering. IOS Press, 2014, pp. 223–232, ISBN 978-1-61499-439-8 (print); 978-1-61499-440-4 (online). [Online]. Available: http://ebooks.iospress.nl/volumearticle/37865

[6] ——, "Lessons learned in participative multidisciplinary design optimization," Journal of Aerospace Operations, 2016, DOI 10.3233/AOP-150054. [Online]. Available: http://content.iospress.com/articles/journal-of-aerospace-operations/aop054

[7] M. Kunde and A. Schreiber, "Advantages of an integrated simulation environment," CEAS Aerospace Aerodynamics Research Conference, Sep. 16-19, 2013, ISBN 789175195193, ISSN: 0001-9240. [Online]. Available: http://www.ceas2013.org/images/images/CEAS2013.pdf

[8] D. Seider, "Open source framework RCE: Integration, automation, collaboration," electronic, Toulouse, France, Nov. 26-28, 2014, presentation. [Online]. Available: http://elib.dlr.de/93323/

[9] E. Moerland et al., "Collaborative aircraft design using an integrated and distributed multidisciplinary product development process," in ICAS 2016, Daejeon, Korea, 2016, forthcoming.

[10] C. Liersch and M. Hepperle, "A distributed toolbox for multidisciplinary preliminary aircraft design," CEAS Aeronautical Journal, vol. 2, pp. 57–68, Dec. 2011, DOI: 10.1007/s13272-011-0024-6. [Online]. Available: https://link.springer.com/article/10.1007%2Fs13272-011-0024-6

[11] B. Nagel et al., "Communication in aircraft design: Can we establish a common language?" in 28th International Congress of the Aeronautical Sciences ICAS 2012, Brisbane, Australia, Sep. 23-28, 2012.

[12] E. Moerland, R.-G. Becker, and B. Nagel, "Collaborative understanding of disciplinary correlations using a low-fidelity physics based aerospace toolkit," CEAS Aeronautical Journal, vol. 6, pp. 441–454, Sep. 2015, DOI: 10.1007/s13272-015-0153-4.

[13] I. Zioni, "What is DevOps? The beginner's guide," electronic, Aug. 17, 2016. [Online]. Available: https://dzone.com/articles/what-is-devops-the-beginners-guide-from-logzio

[14] M. Hüttermann, "DevOps matrix clarifies areas of DevOps practice," electronic, Jun. 25, 2014. [Online]. Available: https://www.sdxcentral.com/articles/contributed/devops-matrix-clarifies-devops-practice-michael-huettermann/2014/06/

[15] L. Prechelt, "An empirical comparison of seven programming languages," Computer, vol. 33, Oct. 2000, ISSN: 0018-9162; DOI: 10.1109/2.876288. [Online]. Available: https://dl.acm.org/citation.cfm?id=621567

[16] H.-C. Fjeldberg, "Polyglot programming. a business perspective," Master's thesis, Norwegian University of Science and Technology, Department of Computer and Information Science, 2008.

[17] F. C. A. Tomassetti, "Polyglot software development," Ph.D. dissertation, Politecnico di Torino, 2014. [Online]. Available: http://porto.polito.it/2537697/

[18] H. Abelson, G. J. Sussman, and J. Sussman, "Structure and interpretation of computer programs," manual, 1990, ISBN 0-262-01153-0 (hardcover), 0-262-51087-1 (paperback). [Online]. Available: https://mitpress.mit.edu/sicp/full-text/sicp/book/book.html

[19] A. C. Rasmussen et al., "Tritonsort: A balanced large-scale sorting system," in Proceedings of the 8th USENIX Symposium on Networked Systems Design and Implementation (NSDI 2011), Boston, MA, Mar. 30-Apr. 1, 2011.

[20] A. C. Rasmussen, "I/O-efficient data-intensive computing," Ph.D. dissertation, University of California, San Diego, 2013.

[21] ——, "Unorthodox paths to high performance," electronic, Aug. 21, 2016. [Online]. Available: https://www.infoq.com/presentations/tritonsort-themis

[22] M. Hering, "How to deal with COTS products in a DevOps world," electronic, Jul. 24, 2016. [Online]. Available: https://www.infoq.com/articles/cots-in-devops-world

[23] R. Burstall, "Christopher strachey - understanding programming languages," Higher-Order and Symbolic Computation, vol. 13: 51, pp. 51–55, Apr. 2000, DOI: 10.1023/A:1010052305354, ISSN: 1388-3690 (print), 1573-0557 (online), http://link.springer.com/article/10.1023%2FA%3A1010052305354. [Online]. Available: http://www.cs.cmu.edu/~crary/819-f09/Strachey67.pdf

[24] A. Bachmann, H. Bergmeyer, and A. Schreiber, "Evaluation of aspect-oriented frameworks in python for extending a project with provenance documentation features," The Python Papers, vol. 6, pp. 3: 1–15, 2011, ISSN: 1834-3147, Free online access: http://ojs.pythonpapers.org/index.php/tpp/issue/view/28.

[25] C. Seaton, "Deoptimizing ruby," electronic, Nov. 17, 2014. [Online]. Available: http://chrisseaton.com/rubytruffle/deoptimizing/