ISBN: 978-3-95450-221-9 ISSN: 2226-0358 doi:10.18429/JACoW-ICALEPCS2021-MOPV040

INTRODUCING PYTHON AS A SUPPORTED LANGUAGE FOR ACCELERATOR CONTROLS AT CERN

P. Elson, C. Baldi, I. Sinkarenko, CERN, Geneva, Switzerland

Abstract

In 2019, Python was adopted as an officially supported language for interacting with CERN's accelerator controls. In practice, this change of status was as much pragmatic as it was progressive - Python has been available as part of the underlying operating system for over a decade and unofficial Python interfaces to controls have existed since at least 2015. So one might ask: what really changed when Python adoption became official?

This paper will discuss what it takes to officially support Python in a controls environment and will focus on the cultural and technological shifts involved in running Python operationally. It will highlight some of the infrastructure that has been put in place at CERN to facilitate a stable and user-friendly Python platform, as well as some of the key decisions that have led to Python thriving in CERN's accelerator controls domain. Given its general nature, it is hoped that the approach presented in this paper can serve as a reference for other scientific organisations from a broad range of fields who are considering the adoption of Python in an operational context.

INTRODUCTION

The Python language, and specifically the cPython implementation for which this paper concerns itself, has one of the largest and fastest growing developer communities of any programming language [1]. It has been jokingly described as "the second best programming language for anything" [2] on the basis of its ability to adapt to a broad array of problems, including domains of particular interest to accelerator controls such as system automation, data analytics & machine learning, web services, and graphical user interfaces (GUIs). This flexibility comes less from the "batteries included" Python standard library, and is more a reflection of a rich suite of third-party packages available from the Python Package Index (PyPI).

CERN's Accelerator & Technology sector, referred to as simply "accelerator sector" in this paper, is responsible for the operation and exploitation of the whole accelerator complex, including the LHC, and for the development of new projects and technologies [3]. Python has been in use in the accelerator sector for a number of years, solving a diverse set of problems including fundamental physics simulations, gathering and analysis of data for machine development (MD) studies, rapid prototyping of services, GUI applications, and much more. For high-level operational accelerator controls at CERN, Java was the only supported programming language until the adoption of Python in 2019. The growth of Python has meant that more people are joining CERN with prior Python experience, and the simplicity of the language is compelling for the large user base of domain specialists for whom software development is a tool to achieve a specific task rather than being the main focus of their work.

Promotion of Python to "supported" status includes the provision of a software infrastructure to allow the operational 24/7 running of Python for mission-critical high-level accelerator controls, as well as infrastructure, such as tooling and a support service, to aid with effective development of such applications & services. Much inspiration was drawn from the existing Java service for accelerator controls, though to directly emulate the Java experience in Python would be to suffer the worst of both worlds and to risk missing the advantages of Python adoption. Therefore, a key objective was to preserve the spirit of Python in the service provided.

The objectives of this paper are to outline some common practices observed before the adoption of Python; to highlight some of the idiomatic Python practices that should be preserved; and at a high level, to present some of the infrastructure put in place to facilitate the use of Python in the accelerator sector at CERN. With this, it is hoped that other scientific organisations may be able to identify existing practices and potential areas of improvement in their own journey to bring Python into an operational context.

As a reflection of the computers in use for high-level controls in the accelerator sector, this paper predominantly focuses on x86-64 architecture, variants of Linux operating systems (OS) such as CentOS, and bare-metal deployments to machines which are often multi-purpose and multi-user.

A TYPICAL PYTHON STARTING POINT

Python Already in Use for Many Years

Python isn't new, and is almost certainly in use in most medium to large scientific organisations today. Typically installed by default in Linux-based operating systems, the barrier to first use of Python is relatively low – scripts can be executed directly by the Python interpreter for a rapid and iterative development experience. As they develop, re-use of functions and other definitions by gathering scripts together into a library of modules becomes desirable. Historically, this was achieved by setting the PYTHONPATH environment variable to point to a directory of packages for consideration in the Python import machinery. At the same time, it is common to want to make use of the rich set of 3rd party, open-source software (OSS) Python packages. On a machine dedicated to a particular task, it is likely that the OS package manager will suffice for the most popular libraries, but eventually it is inevitable that one or more of the 300,000+ packages on the Python Package Index (PyPI) [4] is not available, at which point using the Package Installer for Python (pip) becomes essential.

A common solution to minimise the need for user installations is to provide a dedicated, feature complete, software stack, such as the Anaconda Python distribution [5]. Indeed, in the CERN accelerator sector a number of such distributions existed, yet the need to augment the environment with extra packages using pip remained.

Pip and Virtual Environments

Whilst pip is not part of the standard library, it is considered the de-facto package manager for Python and is the basis for a number of other popular Python tools such as tox, venv and poetry. This effectively makes pip an essential tool for everyday Python development.

The pip package provides a command-line tool from which other packages may be installed. The default file location used by pip when installing packages is the same location as where pip itself is installed. As a result, by default the installation of a package using pip requires user permissions of at least those of the user who installed pip in the first place. For Python & pip installed through the OS package manager, it is common, yet highly discouraged [6], to see the use of “sudo pip install” to grant pip sufficient privilege to install the packages into the system prefix. A common alternative solution is to request that pip installs packages into the “user site-packages” location found in the user’s home directory. This location is typically included in Python’s import search path, therefore both solutions result in packages being made globally available by default for all subsequent Python invocations. This means that the behaviour of an entirely independent Python application or service can be accidentally changed, without it itself having been modified. At CERN, since user home directories are on network mounted filesystems, the user site-packages location can impact applications running on different machines under the same user, and this in particular was found to be a common source of application instability.

The popular approach to avoid the global effect of pip installation is to use a virtual environment. Virtual environments can be created with the standard library “venv” package. A virtual environment is a prefix directory containing a link back to a “base” Python executable, and into which Python packages can be installed. In this way it is possible to have many virtual environments for a single base Python installation. When pip is installed in a virtual environment, packages can be installed without a global effect upon other Python invocations, and the environment is therefore considered “isolated”. By design, it is possible for virtual environments to be created by users who do not have permission to write to the base prefix, thereby giving a clear path to allow a centralised base Python installation which can be extended by users through the use of virtual environments.

RUNNING PYTHON OPERATIONALLY

Advantages of Modules over Scripts

Whilst Python is an interpreted language, and Python script execution is undoubtedly the simplest way of running Python code, there are a number of advantages of running modules rather than executing scripts. When running a script:

- The directory containing the script is put on the Python path, with a higher precedence than any other directory, including that of the standard library. This can easily result in “name shadowing” and accidentally replacing parts of the standard library and third-party packages.
- There is no standard mechanism for sharing scripts (e.g. they can’t be installed with pip), and no mechanism for declaring their dependencies.

A further observation is that scripts tend to encourage big blocks of linear code, with little factorisation and limited code reuse. For example, it is common to see a large directory of unmanaged scripts which each have subtle variations of one another. One desirable aspect of reuse, beyond the obvious ability to maintain functionality in a single place, is the ability to systematically test functionality through tools such as pytest. Whilst possible to write modules with exactly the same limitations as scripts, there is a much clearer consensus that Python modules should avoid the import-time behaviour in this way [7], and as a result, modules gathered into packages tend to be better equipped for effective code re-use.

As part of the guidelines developed for Python in the accelerator sector it was therefore recommended that the creation of packages, which are a collection of modules under a namespace, be favoured over the creation of collections of scripts. The immediate benefit to the user is that code can be easily re-used and tested, with declared dependencies being automatically installed alongside the code. Structurally, it enables relative imports in order to access sibling modules, and it avoids the need for later migration to a package structure as the code evolves.

The efficacy of this recommendation is hard to quantify in general, and strictly enforcing such a policy would be to the detriment of the flexibility that Python brings. Despite this, all known operational Python applications used in the CERN accelerator control room are deployed and run as packages in this way.

The Acc-Py Base Environment

A single, common, software stack containing all Python packages suffers with the problem that the software cannot be updated for one project without risking it breaking for another. When managing such a software stack, either the whole stack must be versioned such that newer iterations can be released without changing existing releases, or packages can be added but practically never removed nor updated. From a developer perspective, the delay to have packages added to the stack inhibits productivity, and as a result, a mechanism to extend the stack locally is essential.

Since virtual environments have become a fundamental part of everyday Python development, they were the preferred approach in the accelerator sector in order to enable pip-based installation of both user-developed and third-party packages.

With the ability to extend a Python distribution through pip, choosing which packages should be in the base distribution is brought into focus. Providing a package in the base distribution has the advantage of guaranteeing that all applications use a consistent version and thereby increasing the chance that tools are able to interoperate. It was concluded that in general this advantage did not guarantee consistency in practice, and that the additional cost of maintaining an extensive base distribution was not justified. As a result, in the accelerator sector base distribution, also known as the Acc-Py base, the decision to include as few packages as possible was made. The packages provided today are limited to those which are hard to pip install in an effective manner, either because they need machine-specific optimisations or specific compilation options. There are currently just two such packages in the Acc-Py base distribution: numpy and PyQt5.

CERN's accelerator operating schedule, typically a 3-4 year physics run followed by a 2-3 year shutdown, requires relatively long periods of stability between upgrade periods. The lifetime of a particular Python version is defined at a maximum of 5 years [8], meaning that a Python version ready to be used from the end of a shutdown period is likely to reach end-of-life before the end of an operational run. This is a key consideration for exposed services, such as web applications, which must, as a result, be re-released on a frequent basis in order to pick up security patches of Python and third-party libraries. The majority of Python usage in CERN's accelerator sector, however, is related to data analysis and GUI applications. For these cases, the biggest impact of Python's end-of-life is that key third-party dependencies will gradually phase out their own support for ageing Python versions, for example Numpy Enhancement Proposal 29 [9] makes clear the intention to limit support within fundamental libraries such as numpy, matplotlib and scikit-learn for older versions of Python. This paints a clear picture that Python is a tool which can be useful for its ease and speed of development, but that there is an additional maintenance cost to be considered over the time frame of CERN's accelerator schedule to ensure that an application or service is adapted as new versions of Python and third-party libraries are released.

In order to facilitate this adaptation to newer Python versions, the Acc-Py base distribution has been designed to allow side-by-side installation with other Acc-Py base versions. Only at the end of an accelerator run will older versions be deprecated and removed, thereby giving application owners the freedom to choose the most appropriate time to adapt their code to a later version. Since no old versions are removed during a run, it can be guaranteed that a correctly isolated application which works at the start of an operational run will continue to work until the end of a run, but no such guarantees can be made for applications across multiple runs.

Whilst possible to install the base distribution on individual machines, the primary mechanism to deliver the distribution to all machines in the accelerator sector is via a shared network mounted disk. One major advantage of this approach is that deployed distributions are the same on all machines, and that updates are instantly available. When compared to a traditional local disk installation, downsides include that the network disk is a single point of failure, and that the added network latency could impact Python start-up time detrimentally. In practice, it has been found that having a single network location is a pragmatic approach and that the consistency across machines is particularly useful.

Tracing Python Startup

To provide an overview of how the various Python distributions are being used, Python start-up logging has been added to the Acc-Py base distributions. For each Python invocation, a centralised log entry is generated containing essential information including username and start-up time, as well as the packages installed in the environment. Currently up to 40,000 logs are received per day, providing a vital window into Python usage, supplying invaluable information when providing support, and highlighting areas of potential improvement to the underlying service.

Adding tracing to an existing service is generally seen unfavourably due to natural fears of an invasion of privacy. Doing so also introduces extra work in the Python startup, this fact also limits what can be reasonably traced. Therefore the general advice for those wishing to add Python startup tracing such as this, is to plan on providing the capability early in the adoption process.

An Internal Package Index and PyPI Proxy

When relying on pip for important work, it is considered good practice to run a PyPI proxy in order to ensure that any downtime of PyPI does not result in loss of productivity, or a loss in ability to deploy applications and services at key moments [10]. Furthermore, the computer network used for accelerator control at CERN prohibits internet access, providing extra motivation for having an internal PyPI proxy. As well as the ability to install public packages from PyPI, it is desirable to allow the installation of internal packages through pip. To solve both of these requirements, a Sonatype Nexus instance was installed. A package repository dedicated to internal releases and a PyPI proxy repository were created. These were then grouped together to form a single user-facing virtual repository.

The fact that Python packages share a single global namespace is a cause of increasingly obscure and often comical package names on PyPI. Whilst the addition of a concept similar to Maven's groupId would help alleviate this, the algorithm used to group together two repositories into one is an important detail not to be overlooked from both a usability and security perspective [11]. In this specific detail, the Python Packaging Authority's devpi package is considered to be the reference implementation, against which Nexus has been found to fall short [12]. This is further exacerbated by the fact that pip itself is able to group together multiple repositories, but does so in an undefined and unexpected manner [13].

Once the package index is set up, pip can be pre-configured to use the package index with no need for further user intervention. For the Acc-Py base distribution and subsequently created virtual environments, configuration is done on a per-installation basis. One subtle and unexpected source of behaviour relating to this is that a suitably configured base distribution does not result in an equivalently configured virtual environment [14]. A patch of venv, the standard library package for virtual environment creation, in the Acc-Py base distribution was needed in order ensure that the configuration was copied into newly created virtual environments.

Along a similar line, the version of pip installed in the base distribution is not the determining factor for the version of pip in a newly created virtual environment. Instead, the version is encoded into the "ensurepip" standard library package, which is fixed at Python release time. Again, further patching of the Acc-Py base distribution was essential to ensure that modern versions of pip are being used in virtual environments.

Development and Deployment Tooling

In order to help users work efficiently and adopt best practice in Python development, a suite of development tools were created. This includes a tool to quickly set up a new Python package, with a well-defined project structure, and a placeholder for tests, sphinx-based documentation and GitLab-based continuous integration configuration.

In addition to the development tooling, a tool to deploy Python applications in a consistent and repeatable manner was developed. The tool, named acc-py-deploy, is comparable to pipx [15] in that applications are considered to be an extension of a Python package which can be installed into isolated virtual environments for later execution. Much like the poetry package [16], acc-py-deploy provides a mechanism to "lock" floating dependency versions into fixed versions. In the case of acc-py-deploy, this includes special behaviour to lock transient Java dependencies being used through JPype [17] such that an application may be deployed consistently as a virtual environment without the risk of dependencies differing between development and production. Deployed applications are run in full isolation mode, with both home-directory-based user site-packages and PYTHONPATH disabled. A unique feature of acc-py-deploy is its integrated ability to elevate to a dedicated service user for the purposes of deployment to a centralised network mounted location. In this way it is possible to have a common application deployment repository, with individuals able to manage the deployment of their own applications, but without needing to grant users direct write access to the installation directory. These features provide a strong guarantee that the application is deployed repeatably and consistently, and that the execution of an application is run in a way which avoids common user or environmental configuration issues.

The uptake of acc-py-deploy has been strong, with over 50 deployed applications in production in its first 12 months. The previously common issue relating to unpredictable application deployment and execution has been eradicated and has been replaced by a reliable and user-friendly operational experience.

PYTHON ADOPTION PRACTICALITIES

Fostering and Supporting a Community

The adoption and centralisation of Python for accelerator controls presents an opportunity to bring together a broad community with a common interest in the Python language and associated tooling. Regular community meetings are therefore arranged in order to provide a space to share news, knowledge and experiences relating to Python. Although the effort to coordinate contributions is relatively high, these meetings have been an invaluable medium for dissemination of key information, and they are an effective way to enhance Python skill across the accelerator sector.

The addition of a centralised support service is a fundamental part of officially adopting Python in the accelerator sector. Support includes resolving operational issues quickly and efficiently, as well as providing guidance to users developing software with Python. Prior experience has shown that providing first-rate and highly individualised guidance through a Python support channel can lead to requests reaching above-and-beyond the scope and capacity of a centralised support service. In an effort to foster a greater sense of community, and to mitigate against over individualising support, a community-centric chat space was created, into which questions and discussion is encouraged. Although a simple solution, the uptake has been excellent, and the sense of an active community comes across strongly. Indeed, it has often been the case that the community have been able to efficiently support one-another directly.

Centralisation of Pre-existing Functionality

When a community of users lack the tools needed to achieve a particular task, and there exists no service from which to request such tools be developed, it is common to see domain-specific solutions being created by users.

Whilst these tools are typically not designed for general purpose use, they provide invaluable insight into specific use cases.

In some cases, these solutions become popular amongst users and should be adopted for centralised support and maintenance. In the accelerator sector, one such tool, PyJapc, was developed by domain specialists in order to conveniently interact with the accelerator control system during machine development (MD) studies. The library is technically interesting as it directly bridges to the Java API for Parameter Control (JAPC) library, yet provides a less general API well-adapted for the use case at hand.

The centralisation of such tooling faces the risk of losing the strong user focus and potentially disenfranchising the original authors if the adoption is not done sensitively and pragmatically. In the case of PyJapc, whilst neither perfect nor general, priority was given to ensuring the existing behaviour was well-maintained, tested, and documented. Only after becoming extremely familiar with the code through bug-fixing and minor enhancements did larger API redesigns become a focus of effort. In turn this respect for what already exists buys credibility when proposing future changes, and it is hoped that this will ultimately lead to improved user uptake of future solutions.

Building-Out New Functionality Quickly

The breadth and depth of existing Java infrastructure for accelerator controls made Python adoption all the trickier, as replicating the infrastructure developed over decades of effort was neither practical nor efficient for the adoption of Python. Instead, emphasis was given to creating packages which bridge to existing implementations written in other languages.

For simple RESTful APIs with OpenAPI or Swagger definitions, openapi-generator was used to generate Python bindings. Some post-processing of generated code was added in order to improve code layout and to programmatically inject type annotations for an improved static analysis experience. Whilst runtime bindings would have been possible due to Python's highly dynamic runtime, it was found that the well-defined types from generated code result in a superior API, with support for convenient static analysis and auto-completion in an integrated development environment (IDE).

When exposing existing Java libraries, the binding technology chosen was the JPype package [17]. JPype integrates tightly with the Java Native Interface (JNI) to start a Java Virtual Machine (JVM) in the same process as the Python interpreter. As a result, data can be passed between Java and Python efficiently without the need to copy data. To improve the developer experience, a tool named stubgenj was created to generate Python type annotations for Java libraries. As a result, it is possible to run static analysis tools and IDE auto-completion when interacting with Java libraries through Python and JPype.

The other binding technology used to build-out capability quickly was PyBind11 [18]. The tool generates Python bindings to C++ libraries, based upon C++ declarations of the desired interface. These declarations can be relatively easily crafted to form intuitive and idiomatic Python APIs. The results of creating bindings through PyBind11 are compiled shared-object modules. In order to maximise compatibility of these modules and to allow installation on multiple Linux operating systems, the manylinux2014 [19] standard was used when publishing Python wheels [20].

CONCLUSION

The formal adoption of Python for accelerator controls at CERN was, on the surface, smooth and relatively seamless. Despite this outward appearance some significant culture shifts were introduced, including the removal of one-size-fits-all Python distributions in favour of virtual environments, the discouragement of script writing in favour of Python package creation, and the introduction of a community-focussed culture through user meetings and a collaborative chat space.

In the accelerator sector Acc-Py has been well-received, and as a result the use of Python is growing, users now have more flexibility to pick the best tools for their job, and most critically of all, the operational stability of Python applications and services has been ensured.

Python and its associated ecosystem are fundamentally open-source, and whilst costing more in the short-term, an effort has been made to report and fix bugs identified with the tools being used. In the long-term this strategy pays off, as it minimises the need for local patches and software bifurcation, and eases the future adoption of newer versions. A by-product of this is better software for all, including for other organisations following their own Python adoption story.

Some of the concepts presented around acc-py-deploy, particularly with regards to stability of execution and repeatability of deployment, can be reasonably compared to the benefits of containerisation. It is considered that the deploy concept from acc-py-deploy will work well as a build step for a container image, and that stable execution could then be handled by the container runtime. This is one particular area of potential future work for Acc-Py.

With key infrastructure in place, the Python adoption phase has now been replaced by a growth phase. Growth of the Python community in terms of number of users and the collective Python skill, and growth in terms of the quality & extent of the libraries provided to interact with CERN's accelerator control system.

ISBN: 978-3-95450-221-9 ISSN: 2226-0358 doi:10.18429/JACoW-ICALEPCS2021-MOPV040

REFERENCES

[1] ZDNet review of Programming languages, https://www.zdnet.com/article/programming-languages-javascript-has-most-developers-but-rust-is-the-fastest-growing/

[2] D. Callahan, PyCon 2018 keynote, https://youtu.be/ITksU31c1WY?t=420

[3] Accelerator & Technology Sector, https://ats.web.cern.ch

[4] Python Package Index, https://pypi.org

[5] Anaconda distribution, https://www.anaconda.com/products/individual

[6] pip permissions overview, https://github.com/pypa/pip/issues/1668

[7] Imports should be "as free of side effects as possible", https://realpython.com/python-import

[8] Ł. Langa, PEP 602 - Annual Release Cycle for Python, https://www.python.org/dev/peps/pep-0602

[9] T. Caswell et al., Recommend Python and NumPy version support, https://numpy.org/neps/nep-0029-deprecation_policy.html

[10] H. Schlawack, Recommendation to run a private PyPI mirror, https://hynek.me/talks/python-deployments

[11] What is a dependency confusion attack?, https://secureteam.co.uk/news/what-is-a-dependency-confusion-attack

[12] Disclosure of dependency confusion vulnerability in Nexus for PyPI proxies, https://issues.sonatype.org/browse/NEXUS-24870

[13] Discussion of pip's index group strategy, https://discuss.python.org/t/dependency-notation-including-the-index-url/5659

[14] pip configuration not inherited in virtual environments, https://github.com/pypa/pip/issues/9752

[15] Install and Run Python Applications in Isolated Environments, https://github.com/pypa/pipx

[16] Python dependency management and packaging made easy, https://github.com/python-poetry/poetry

[17] JPype, a module to provide full access to Java from within Python, https://jpype.readthedocs.io

[18] Seamless operability between C++11 and Python, https://pybind11.readthedocs.io

[19] P. Moore, PEP 599 - The manylinux2014 Platform Tag, https://www.python.org/dev/peps/pep-0599

[20] N. Coghlan, PEP 427 - The Wheel Binary Package Format, https://www.python.org/dev/peps/pep-0427