`davos`: The Python package smuggler

Paxton C. Fitzpatrick, Jeremy R. Manning*

Department of Psychological and Brain Sciences
Dartmouth College, Hanover, NH 03755

Abstract

TODO: clean up after writing body

Reproducible code plays many important roles in modern scientific research: it enables scientists to collaborate on projects, replicate and extend prior work, and teach students new concepts and techniques via hands-on, interactive tutorials. However, existing tools that facilitate creating, sharing, and running reproducible code are often highly complex and resource-intensive, creating barriers to both contributing to and benefitting from open science resources. Here, we introduce `davos`, a Python package that allows users to create and share reproducible workflows...

Keywords: Reproducibility, Open science, Python, Jupyter Notebook, Google Colaboratory, Package management

*Corresponding author

Email address: `Jeremy.R.Manning@Dartmouth.edu` (Jeremy R. Manning)

Required Metadata

Current code version

Nr.	**Code metadata description**	**Please fill in this column**
C1	Current code version	v0.1.1
C2	Permanent link to code/repository used for this code version	https://github.com/ContextLab/davos/tree/v0.1.1
C3	Code Ocean compute capsule	
C4	Legal Code License	MIT
C5	Code versioning system used	git
C6	Software code languages, tools, and services used	Python, JavaScript, PyPI/pip, IPython, Jupyter, Ipykernel, PyZMQ. Additional tools used for tests: pytest, Selenium, Requests, mypy, GitHub Actions
C7	Compilation requirements, operating environments & dependencies	Dependencies: Python>=3.6, packaging, setuptools. Supported OSes: MacOS, Linux, Unix-like. Supported IPython environments: Jupyter notebooks, JupyterLab, Google Colaboratory, Binder, IDE-based notebook editors.
C8	If available Link to developer documentation/manual	https://github.com/ContextLab/davos#readme
C9	Support email for questions	contextualdynamics@gmail.com

Table 1: Code metadata

1. Motivation and significance

Modern scientific research frequently entails writing software code for a wide variety of purposes throughout the scientific process. Researchers across disciplines may design and implement complex experiments; collect, store, and analyze large datasets; create visualizations for presentations and publications; and share their findings and techniques with peers, students, and the broader public through tutorials, demos, workshops, and classes. However, one fundamental requirement of virtually any form of research-related code is that its behavior and outputs remain consistent and predictable, no matter when, where, or by whom it is run. This stability can be crucial, for example, to ensuring that data are collected under the same conditions (e.g., across recordings, subjects, or physical locations) over multiple months or years, and that they can be accessed, processed, and analyzed consistently by a research team that may be spread across multiple institutions or countries. Additionally, modern open science practices encourage publicly sharing research code and data so that others may explore, reproduce, learn from, and build upon existing work. Much of the benefit afforded by freely available research code depends on users' ability to execute it and achieve the same result as its original author.

Python [1] has become one of the most widely used and fastest-growing scientific programming languages, in part by combining an accessible, high-level syntax with a rich ecosystem of powerful third-party tools that facilitate rapid development and collaboration [2]. The Python ecosystem offers an extensive data science toolkit with platforms for interactive programming (e.g., Project Jupyter [3], Google Colaboratory), community-maintained libraries for data manipulation (e.g., NumPy [4], SciPy [5], Pandas [6]) and visualization (e.g., Matplotlib [7], seaborn [8]), frameworks for training complex machine learning models (e.g., scikit-learn [9], TensorFlow [10], Hugging Face [11]), and myriad other resources. However, this heavy emphasis on third-party libraries also presents a challenge to writing and sharing stable, reproducible scientific Python code: different versions of the same library may behave differently, adopt changes in syntax, expose different functions and interfaces, add or drop support for specific hardware or software, write (or expect to read) files in different formats, fix (or introduce) bugs, and so on. While these issues exist to some extent in any software language or ecosystem, they have a particular impact on the Python community due to its unusually rapid growth relative to other languages. Ensuring Python code behaves consistently over time and across users therefore typically requires ensuring it is always run with the same specific set of versions for each third-party package used.

One common approach to solving this problem is to create containerized or virtualized Python environments (e.g., using Docker [12], Singularity [13], or conda [14]) tailored to individual applications. A researcher may build such an environment around a particular experiment or analysis pipeline, and exclusively run their code from inside it, "entering" and "exiting" the environment before and after each time they do so. They can also distribute their custom environment alongside their code as a configuration file that explicitly lists required package versions, enabling others to build identical copies for themselves. This allows research teams to deploy experiments on multiple machines for more efficient data collection, collaborate on analyses without introducing conflicts or inconsistencies, and publicly share their study designs and results for others to reproduce, replicate, or adapt to study new questions in the future.

While often effective, this approach bears two notable drawbacks. First, it can add substantially to the technical knowledge, computing resources, and initial setup needed to run or share the actual code of interest. For example, sharing code for an analysis or tutorial that relies on a particular Docker image to run properly would of course necessitate writing and distributing extra configuration files and setup instructions. But far more burdensomely, it also requires that anyone who may want to run the code (in addition to the author seeking to share it!) first be able to install and navigate additional software that is likely far more complex and resource-intensive than the actual analysis or tutorial code it facilitates running. This can introduce a need for both a degree of computer literacy and computational resources that may not be universally accessible, particularly to students or other early-career scientists hoping to learn from publicly available tutorials. These added prerequisites clash with the simplicity and accessibility that have contributed to Python's popularity, and can create significant barriers to both contributing to and taking advantage of open science resources.

Second, while many existing tools allow users to initially populate a Python environment with a fixed set of packages and package versions (e.g., from a `requirements.txt`, `pyproject.toml`, `environment.yml`, `Pipfile`, `Dockerfile RUN` instruction, etc.), few, if any, ensure that these specified requirements *remain* satisfied after they are first installed. The ability to modify an environment after its creation is useful in many cases (e.g., to install additional software, when needed). However, this also makes it easy to inadvertently alter existing packages, potentially leading to subtle issues with code that relies on them. For instance, suppose a researcher has implemented a series of analyses using version 1.0 of "Package X," and decides to perform an additional analysis that requires installing "Package Y." If Package Y depends on version 0.9 of Package X, then Package X will be downgraded to accommodate this new requirement, potentially altering or breaking previous analyses when they are rerun later, either by the researcher or someone with whom they've shared their code. Further, if some analyses require Package Y while others rely on features of Package X not implemented until version 1.0, it's unclear which version the researcher should install in their environment.

The `davos` package provides a novel, Python-native framework for creating reproducible workflows that was designed to address each of these issues. `davos` allows users to specify dependencies directly within the code that...

2. Software description

2.1. Software architecture

The davos package is structured as two subpackages: a set of "core" modules that implement...

2.2. Software functionalities

2.2.1. The smuggle statement

Importing davos enables an additional Python keyword: "smuggle". The smuggle statement can be used as a drop-in replacement for Python's built-in import statement to load libraries, modules, and other objects into the current namespace. However, whereas import will fail if the requested package is not installed locally, smuggle statements can handle missing packages on the fly. If a smuggled package does not exist in the local environment, davos will install it, expose its contents to Python's import machinery, and load it into the namespace for immediate use.

2.2.2. The onion comment

For greater control over the behavior of smuggle statements, davos defines an additional construct called the *onion comment*. An onion comment is a special type of inline comment that may be placed on a line containing a smuggle statement to customize how davos searches for the smuggled package locally and, if necessary, how it should be installed. Onion comments follow a simple syntax based on the "type comment" syntax introduced in PEP 484 [18] and are designed to make managing packages via davos intuitive and familiar. To construct an onion comment, simply provide the name of the installer program (e.g., pip) and the same arguments one would use to install the package as desired manually via the command line (see Fig. 1).

```
import davos

# if numpy is not installed locally, pip-install it and display verbose output
smuggle numpy as np    # pip: numpy --verbose

# pip-install pandas without using or writing to the package cache
smuggle pandas as pd    # pip: pandas --no-cache-dir

# install scipy from a relative local path, in editable mode
from scipy.stats smuggle ttest_ind    # pip: -e ../../pkgs/scipy
```

Figure 1: **Figure 1**

2.2.3. The `davos` config

2.2.4. Additional functionality

2.3. Sample code snippets analysis (optional)

3. Illustrative Examples

4. Impact

Since its initial release, `davos` has found use in a variety of applications. In addition to managing data analysis environments for multiple ongoing research studies, `davos` is being used by both students and instructors in programming courses such as *Storytelling with Data* [19] (an open course on data science, visualization, and communication) to simplify distributing lessons and submitting assignments, as well as in online demos such as `abstract2paper` [20] (an example application of GPT-Neo) to share ready-to-run code that installs dependencies automatically.

5. Conclusions

Author Contributions

Paxton C. Fitzpatrick: Conceptualization, Methodology, Software, Validation, Writing - Original Draft, Visualization. **Jeremy R. Manning**: Conceptualization, Resources, Writing - Review & Editing, Supervision, Funding acquisition.

Funding

Our work was supported in part by NSF grant number 2145172 to J.R.M. The content is solely the responsibility of the authors and does not necessarily represent the official views of our supporting organizations.

Declaration of Competing Interest

We wish to confirm that there are no known conflicts of interest associated with this publication and there has been no significant financial support for this work that could have influenced its outcome.

Acknowledgements

References

[1] G. van Rossum, Python reference manual, Department of Computer Science [CS] (R 9525) (1995).

[2] E. Muller, J. A. Bednar, M. Diesmann, M.-O. Gewaltig, M. Hines, A. P. Davison, Python in neuroscience, Frontiers in Neuroinformatics 9 (2015) 11. doi:10.3389/fninf.2015.00011.

[3] T. Kluyver, B. Ragan-Kelley, F. Pérez, B. Granger, M. Bussonnier, J. Frederic, K. Kelley, J. Hamrick, J. Grout, S. Corlay, P. Ivanov, D. Avila, S. Abdalla, C. Willing, Jupyter notebooks – a publishing format for reproducible computational workflows., in: F. Loizides, B. Scmidt (Eds.), Positioning and Power in Academic Publishing: Players, Agents and Agendas, IOS Press, Netherlands, 2016, pp. 97–90. doi:10.3233/978-1-61499-649-1-87.

[4] C. R. Harris, K. J. Millman, S. J. van der Walt, R. Gommers, P. Virtanen, D. Cournapeau, E. Wieser, J. Taylor, S. Berg, N. J. Smith, R. Kern, M. Picus, S. Hoyer, M. H. van Kerkwijk, M. Brett, A. Haldane, J. F. del Río, M. Wiebe, P. Peterson, P. Gérard-Marchant, K. Sheppard, T. Reddy, W. Weckesser, H. Abbasi, C. Gohlke, T. E. Oliphant, Array programming with NumPy, Nature 585 (7825) (2020) 357–362. doi:10.1038/s41586-020-2649-2.

[5] P. Virtanen, R. Gommers, T. E. Oliphant, M. Haberland, T. Reddy, D. Cournapeau, E. Burovski, P. Peterson, W. Weckesser, J. Bright, S. J. van der Walt, M. Brett, J. Wilson, K. J. Millman, N. Mayorov, A. R. J. Nelson, E. Jones, R. Kern, E. Larson, C. J. Carey, İ. Polat, Y. Feng, E. W. Moore, J. VanderPlas, D. Laxalde, J. Perktold, R. Cimrman, I. Henriksen, E. A. Quintero, C. R. Harris, A. M. Archibald, A. H. Ribeiro, F. Pedregosa, P. van Mulbregt, SciPy 1.0 Contributors, SciPy 1.0: Fundamental Algorithms for Scientific Computing in Python, Nature Methods 17 (3) (2020) 261–272. doi:10.1038/s41592-019-0686-2.

[6] W. McKinney, Data structures for statistical computing in Python, in: Proceedings of the Python in Science Conference, 2010, pp. 51–56. doi:10.25080/Majora-92bf1922-00a.

[7] J. D. Hunter, Matplotlib: A 2D graphics environment, Computing in Science and Engineering 9 (3) (2007) 90–95. doi:10.1109/MCSE.2007.55.

[8] M. L. Waskom, seaborn: statistical data visualization, Journal of Open Source Software 6 (60) (2021) 3021. doi:10.21105/joss.03021.

[9] F. Pedregosa, G. Varoquaux, A. Gramfort, V. Michel, B. Thirion, O. Grisel, M. Blondel, P. Prettenhofer, R. Weiss, V. Dubourg, J. Vanderplas, A. Passos, D. Cournapeau, M. Brucher, M. Perrot, E. Duchesnay, Scikit-learn: machine learning in Python, Journal of Machine Learning Research 12 (2011) 2825–2830.

[10] M. Abadi, A. Agarwal, P. Barham, E. Brevdo, Z. Chen, C. Citro, G. S. Corrado, A. Davis, J. Dean, M. Devin, S. Ghemawat, I. Goodfellow, A. Harp, G. Irving, M. Isard, Y. Jia, R. Jozefowicz, L. Kaiser, M. Kudlur, J. Levenberg, D. Mané, R. Monga, S. Moore, D. Murray, C. Olah, M. Schuster, J. Shlens, B. Steiner, I. Sutskever, K. Talwar, P. Tucker, V. Vanhoucke, V. Vasudevan, F. Viégas, O. Vinyals, P. Warden, M. Wattenberg, M. Wicke, Y. Yu, X. Zheng, TensorFlow: Large-scale machine learning on heterogeneous systems, software available from tensorflow.org (2015).
URL https://www.tensorflow.org/

[11] T. Wolf, L. Debut, V. Sanh, J. Chaumond, C. Delangue, A. Moi, P. Cistac, C. Ma, Y. Jernite, J. Plu, C. Xu, T. Le Scao, S. Gugger, M. Drame, Q. Lhoest, A. M. Rush, Transformers: State-of-the-Art Natural Language Processing, Association for Computational Linguistics, 2020, pp. 38–45.
URL https://www.aclweb.org/anthology/2020.emnlp-demos.6

[12] D. Merkel, Docker: lightweight linux containers for consistent development and deployment, Linux Journal 239 (2) (2014) 2.

[13] G. M. Kurtzer, V. Sochat, M. W. Bauer, Singularity: Scientific containers for mobility of compute, PLoS One 12 (5) (2017) e0177459.

[14] Anaconda, Inc., conda, https://docs.conda.io (2012).

[15] F. Pérez, B. E. Granger, Ipython: a system for interactive scientific computing, Computing in science & engineering 9 (3) (2007) 21–29. doi:10.1109/MCSE.2007.53.

[16] B. Granger, J. Grout, Jupyterlab: Building blocks for interactive computing, Slides of presentation made at SciPy (2016).

[17] B. Ragan-Kelley, C. Willing, Binder 2.0-reproducible, interactive, sharable environments for science at scale, in: F. Akici, D. Lippa, D. Niederhut, , M. Pacer (Eds.), Proceedings of the 17th Python in Science Conference, 2018, pp. 113–120. doi:10.25080/MAJORA-4AF1F417-011.

[18] G. van Rossum, J. Lehtosalo, Ł. Langa, Type Hints, PEP 484 (September 2014).
URL https://www.python.org/dev/peps/pep-0484

[19] J. R. Manning, Storytelling with Data (June 2021). doi:10.5281/zenodo.5182775.
URL https://doi.org/10.5281/zenodo.5182775

[20] J. R. Manning, abstract2paper (2021).
URL https://github.com/ContextLab/abstract2paper