

# davos: a Python package “smuggler” for constructing lightweight reproducible notebooks

Paxton C. Fitzpatrick, Jeremy R. Manning\*

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

---

## Abstract

Reproducibility is a core requirement of modern scientific research. For computational research, reproducibility means that code should produce the same results, even when run on different systems. A standard approach to ensuring reproducibility entails packaging a project’s dependencies along with its primary code base. Existing solutions vary in how deeply these dependencies are specified, ranging from virtual environments, to containers, to virtual machines. Each of these existing solutions requires installing or setting up a system for running the desired code, increasing the complexity and time cost of sharing or engaging with reproducible science. Here, we propose a lighter-weight solution: the **davos** library. When used in combination with a notebook-based Python project, **davos** provides a mechanism for specifying (and automatically installing) the correct versions of the project’s dependencies. The **davos** library further ensures that those packages and specific versions are used every time the notebook’s code is executed. This enables researchers to share a complete reproducible copy of their code within a single Jupyter notebook file.

*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	Metadata value
C1	Current code version	v0.1.1
C2	Permanent link to code/repository used for this code version	<a href="https://github.com/ContextLab/davos/tree/v0.1.1">https://github.com/ContextLab/davos/tree/v0.1.1</a>
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, and dependencies	Dependencies: Python $\geq 3.6$ , packaging, setuptools. Supported OSes: MacOS, Linux, Unix-like. Supported IPython environments: Jupyter notebooks, JupyterLab, Google Colaboratory, Binder, IDE-based notebook editors.
C8	Link to developer documentation/manual	<a href="https://github.com/ContextLab/davos#readme">https://github.com/ContextLab/davos#readme</a>
C9	Support email for questions	contextualdynamics@gmail.com

Table 1: Code metadata

## 1. Motivation and significance

The same computer code may not behave identically under different circumstances. For example, when code depends on external libraries, different versions of those libraries may function differently. Or when CPU or GPU instruction sets differ across machines, the same high-level code may be compiled into different machine instructions. Because executing identical code does not guarantee identical outcomes, code sharing alone is often insufficient for enabling researchers to reproduce each other’s work, or to collaborate on projects involving data collection or analysis.

Within the Python [1] community, external packages that are published in the most popular repositories [2, 3] are associated with version numbers and

12 tags that allow users to guarantee they are installing exactly the same code  
13 across different computing environments [4]. While it is *possible* to manually  
14 install the intended version of every dependency of a Python script or pack-  
15 age, manually tracking down those dependencies can impose a substantial  
16 burden on the user and create room for mistakes and inconsistencies. Fur-  
17 ther, when dependency versions are left unspecified, replicating the original  
18 computing environment becomes difficult or impossible.

19 Computational researchers and other programmers have developed a broad  
20 set of approaches and tools to facilitate code sharing and reproducible out-  
21 comes (Fig. 1). At one extreme, simply distributing a set of Python scripts  
22 (`.py` files) may enable others to use or gain insights into the relevant work.  
23 Because Python is installed by default on most modern operating systems,  
24 for some projects, this may be sufficient. Another popular approach entails  
25 creating Jupyter notebooks [8] that comprise a mix of text, executable code,  
26 and embedded media. Notebooks may call or import external scripts or  
27 libraries—even intersperse snippets of other programming or markup lang-  
28 uages—in order to provide a more compact and readable experience for users.  
29 Both of these systems (Python scripts and notebooks) provide a convenient  
30 means of sharing code, with the caveat that they do not specify the comput-  
31 ing environment in which the code is executed. Therefore the functionality of  
32 code shared using these systems cannot be guaranteed across different users  
33 or setups.

34 At another extreme, virtual machines [9, 10, 11] provide a hardware-level  
35 simulation of the desired system. Virtual machines are typically isolated such  
36 that installing or running software on a virtual machine does not impact the  
37 user’s primary operating system or computing environment. Containers [e.g.,  
38 12, 13] provide a similar “isolated” experience. Although containerized envi-  
39 ronments do not specify hardware-level operations, they are typically pack-  
40 aged with a complete operating system, in addition to a complete copy of  
41 Python and any relevant package dependencies. Virtual environments [e.g.,  
42 6, 7] also provide a computing environment that is largely separated from the  
43 user’s main environment. They incorporate a copy of Python and the target  
44 software’s dependencies, but virtual environments do not specify or repro-  
45 duce an operating system for the runtime environment. Each of these systems  
46 (virtual machines, containers, and virtual environments) guarantees (to dif-  
47 fering degrees—at the hardware level, operating system level, and Python  
48 environment level, respectively) that the relevant code will run similarly for  
49 different users. However, each of these systems also relies on additional soft-  
50 ware that can be complex or resource-intensive to install and use, creating  
51 potential barriers to both contributing to and taking advantage of open sci-  
52 ence resources.

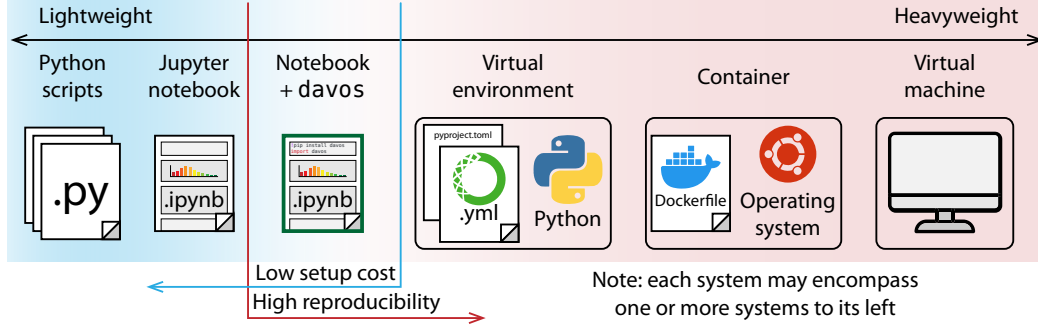


Figure 1: **Systems for sharing code within the Python ecosystem.** From left to right: plain-text **Python scripts** (.py files) provide the most basic “system” for sharing raw code. Scripts may reference external libraries, but those libraries must be manually installed on other users’ systems. Further, any checking needed to verify that the correct versions of those libraries were installed must also be performed manually. **Jupyter notebooks** (.ipynb files) comprise embedded text, executable code, and media (including rendered figures, code output, etc.). When the **davos** library is imported into a Jupyter notebook, the notebook’s functionality is extended to automatically install any required external libraries (at their correct versions, when specified). **Virtual environments** allow users to install an isolated copy of Python and all required dependencies. This typically entails distributing a configuration file (e.g., a `pyproject.toml` [5] or `environment.yml`) that specifies all project dependencies (including version numbers of external libraries) alongside the primary code base. Users can then install a third-party tool [e.g., 6, 7] to read the file and build the environment. **Containers** provide a means of defining an isolated environment that includes a complete operating system (independent of the user’s operating system), in addition to (optionally) specifying a virtual environment or other configurations needed to provide the necessary computing environment. Containers are typically defined using specification files (e.g., a plain-text `Dockerfile`) that instruct the virtualization engine regarding how to build the containerized environment. **Virtual machines** provide a complete hardware-level simulation of the computing environment. In addition to simulating specific hardware, virtual machines (typically specified using binary images files) must also define operating system-level properties of the computing environment. Systems to the left of the blue vertical line entail sharing individual files, with no additional installation or configuration needed to run the target code. Systems to the right of the red vertical line support precise control over dependencies and versioning. Notebooks enhanced using the **davos** library are easily shareable and require minimal setup costs, while also facilitating high reproducibility by enabling precise control over project dependencies.

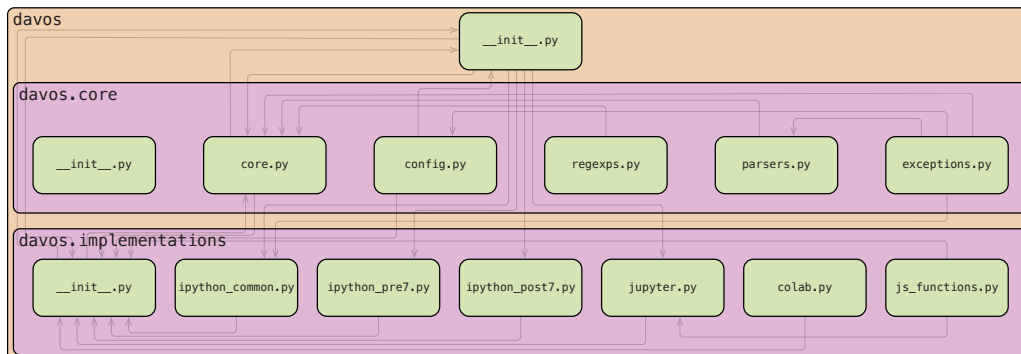


Figure 2: **Package structure.**

We designed **davos** to occupy a “sweet spot” between these extremes. **davos** is a notebook-installable package that adds functionality to the default notebook experience. Like standard Jupyter notebooks, **davos**-enhanced notebooks allow researchers to include text, executable code, and media within a single file. No further setup or installation is required, beyond what is needed to run standard Jupyter notebooks. And like virtual environments, **davos** provides a convenient mechanism for fully specifying (and installing, as needed) a complete set of Python dependencies, including package versions.

## 2. Software description

The **davos** package is named after Davos Seaworth, a smuggler often referred to as “the Onion Knight” from the series *A Song of Ice and Fire* by George R. R. Martin.

### 2.1. Software architecture

The **davos** package consists of two interdependent subpackages (see Fig. 2). The first, **davos.core**, comprises a set of modules that implement the bulk of the package’s core functionality, including pipelines for installing and validating packages, custom parsers for the **smuggle** statement (see Section 2.2.1) and onion comment (see Section 2.2.2), and a runtime interface for configuring **davos**’s behavior (see Section 2.2.3). However, certain critical aspects of this functionality require (often substantially) different implementations depending on properties of the notebook environment in which **davos** is used (e.g., whether the frontend is provided by Jupyter or Google Colaboratory, or which version of IPython [14] is used by the notebook kernel). To deal with this, environment-dependent parts of core features and behaviors are isolated and abstracted to “helper functions” in the **davos.implementations**

subpackage. This second subpackage defines multiple, interchangeable versions of each helper function, organized into modules by the conditions that trigger their use. At runtime, `davos` detects various features in the notebook environment and selectively imports a single version of each helper function into the top-level `davos.implementations` namespace, allowing `davos.core` modules to access the correct implementations for the current notebook environment in a single, consistent location. An additional benefit of this design pattern is that it allows maintainers or users to easily extend `davos` to support new, updated, or custom notebook variants by creating a new `davos.implementations` module with any necessary tweaks to the existing helper functions.

## 2.2. Software functionalities

### 2.2.1. The `smuggle` statement

Importing `davos` in an IPython notebook appears to enable an additional Python keyword: “`smuggle`” (see Section 2.3 for details on how this works). 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 automatically, 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, downloads and installs it. Onion comments follow a simple format based on the “type comment” syntax introduced in PEP 484 [15], and are designed to make managing packages with `davos` intuitive and familiar. To construct an onion comment, users provide the name of the installer program (e.g., `pip`) and the same arguments one would use to manually install the package as desired via the command line:

```

# enable smuggle statements
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

```

Occasionally, a package's distribution name (i.e., the name used when installing it) may differ from its top-level module name (i.e., the name used when importing it). In such cases, an onion comment can be used to ensure **davos** installs the proper distribution if the smuggled package can't be found locally:

```

# package is named "python-dateutil" on PyPI, but imported as "dateutil"
smuggle dateutil    # pip: python-dateutil

# package is named "scikit-learn" on PyPI, but imported as "sklearn"
from sklearn.decomposition smuggle PCA    # pip: scikit-learn

```

However, the most powerful use of the onion comment is making **smuggle** statements *version-sensitive*. If an onion comment includes a version specifier [4], **davos** will ensure that the version of the package loaded into the notebook matches the specific version requested, or satisfies the given version constraints. If the smuggled package exists locally, **davos** will extract its version info from its metadata and compare it to the specifier provided. If the two are incompatible (or no local installation is found), **davos** will install and load a suitable version of the package instead:

```

# specifically use matplotlib v3.4.2, pip-installing it if needed
smuggle matplotlib.pyplot as plt    # pip: matplotlib==3.4.2

# use a version of seaborn no older than v0.9.1, but before v0.11
smuggle seaborn as sns    # pip: seaborn>=0.9.1,<0.11

```

Onion comments can also be used to smuggle specific VCS references (e.g., Git [16] branches, commits, tags, etc.):

```

# use quail as the package existed on GitHub at commit 6c847a4
smuggle quail    # pip: git+https://github.com/ContextLab/quail.git@6c847a4

```

130 **davos** processes onion comments internally before forwarding arguments to  
131 the installer program. In addition to preventing onion comments from being  
132 used as a vehicle for shell injection attacks, this allows **davos** to take certain  
133 logical actions when particular arguments are passed. For example, the **-I/**  
134 **--ignore-installed**, **-U/--upgrade**, and **--force-reinstall** flags will all  
135 cause **davos** to skip searching for a smuggled package locally before installing  
136 a new copy:

```
# install hypertools v0.7 without first checking for it locally
smuggle hypertools as hyp      # pip: hypertools==0.7 --ignore-installed

# always install the latest version of requests, including pre-releases
from requests smuggle Session  # pip: requests --upgrade --pre
```

137

138 Similarly, passing **--no-input** will temporarily enable **davos**'s non-interactive  
139 mode (see Section 2.2.2), and installing a smuggled package into **<dir>** with  
140 **--target <dir>** will cause **dir** to be prepended to the module search path  
141 (**sys.path**), if necessary, so the package can be imported

### 142 2.2.3. The *davos* config

143 The **davos** config provides a simple, high-level interface for controlling  
144 various aspects of **davos**'s behavior. After importing **davos**, the **davos.config**  
145 object (a singleton) exposes a number of configurable options as attributes  
146 that can be assigned different values, checked at runtime, and displayed in  
147 the notebook cell output (see Fig. INSERT REF TO ILLUSTRATIVE EX-  
148 AMPLE FIG for example usage). These include:

- 149 • **.active**: This option allows users to disable **davos** functionality (i.e.,  
150 support for **smuggle** statements and onion comments) for subsequent  
151 notebook cells by setting its value to **False**. **davos** can be re-enabled  
152 at any time by setting this option to **True** (the default when **davos** is  
153 first imported). See Section 2.3 for additional info.
- 154 • **.auto\_rerun**: This option controls how **davos** behaves when attempt-  
155 ing to **smuggle** a new version of a package that was previously imported  
156 and cannot be reloaded. This can happen if the package includes exten-  
157 sion modules that dynamically link C or C++ objects to the Python  
158 interpreter itself, and the code that generates those objects was changed  
159 between the old and new versions. If this option is set to **True**, **davos**  
160 will automatically restart the notebook kernel and rerun all code up to  
161 (and including) the current **smuggle** statement. If **False** (the default),  
162 **davos** will instead issue a warning, pause execution, and prompt the



163 user with buttons to either restart and rerun the notebook, or con-  
 164 tinue running with the previously imported package version until the  
 165 next time the kernel is restarted manually. (Note: not configurable in  
 166 Google Colaboratory).

- 167 • `.confirm_install`: If `True` (default: `False`), `davos` will require user  
 168 confirmation (`[y]es/[n]o` input) before installing a smuggled package.
- 169 • `.noninteractive`: Setting this option to `True` (default: `False`) en-  
 170 ables non-interactive mode, in which all user input and confirmation  
 171 is disabled. Note that in non-interactive mode, the `confirm_install`  
 172 option is set to `False`, and if `auto_rerun` is `False`, `davos` will throw an  
 173 error if a smuggled package cannot be reloaded, rather than prompting  
 174 the user.
- 175 • `.pip_executable`: This option’s value specifies the path to the `pip`  
 176 executable used to install smuggled packages. The default is program-  
 177 matically determined from the Python environment and falls back to  
 178 `sys.executable -m pip` if no executable can be found.
- 179 • `.suppress_stdout`: If `True` (default: `False`), suppress all unnecessary  
 180 output issued by both `davos` and the installer program. This can be  
 181 useful when smuggling packages that need to install many dependencies  
 182 and/or generate extensive output. If the installer program throws an  
 183 error, both the `stdout` and `stderr` streams will be displayed with the  
 184 traceback.

185 The top-level `davos` namespace also defines a handful of convenience func-  
 186 tions for setting and checking `davos`’s active/inactive state (`davos.activate()`;  
 187 `davos.deactivate()`; `davos.is_active()`) as well as the `davos.configure()`  
 188 function, which allows setting multiple config options at once.

### 189 *2.3. Implementation details*

190 Functionally, importing `davos` appears to define “`smuggle`” as a Python  
 191 keyword, similar to “`import`”, “`def`”, or “`return`”. It also appears to cause  
 192 comments to be parsed, and their contents potentially able to affect code  
 193 behavior, which they normally are not. However, `davos` doesn’t actually  
 194 modify the rules of Python’s parser or lexical analyzer—in fact, modifying  
 195 the Python grammar isn’t possible at runtime, as doing so would require  
 196 rebuilding the interpreter. Instead, `davos` leverages the IPython notebook  
 197 backend to implement the `smuggle` statement and onion comment via a com-  
 198 bination of namespace injections and its own (far simpler) custom parser.

199 The `smuggle` keyword can be enabled and disabled at any time by “ac-  
200 tivating” and “deactivating” `davos` (see Section 2.2.3, above). When `davos`  
201 is first imported, it is activated automatically. Activating `davos` triggers  
202 two actions: (1) the `smuggle()` function is injected into the IPython user  
203 namespace, and (2) the `davos` parser is registered as a custom IPython input  
204 transformer. IPython preprocesses all executed code as plain text before it is  
205 sent to the Python parser, in order to handle special constructs like `%magic`  
206 and `!shell` commands. `davos` hooks into this process to transform `smuggle`  
207 statements into syntactically valid Python code. The `davos` parser uses a  
208 complex regular expression [17] to match lines of code containing `smuggle`  
209 statements (and, optionally, onion comments), extract relevant information  
210 from their text, and replace them with equivalent calls to the `smuggle()`  
211 function. For example, if a user runs a notebook cell containing

```
212 smuggle numpy as np      # pip: numpy>1.16,<=1.20 -vv
```

213 the code that is actually executed by the Python interpreter would be

```
214 smuggle(name="numpy", as_="np", installer="pip",  
        args_str="\"numpy>1.16,<=1.20 -vv\"",  
        installer_kwargs={'editable': False,  
                          'spec': 'numpy>1.16,<=1.20',  
                          'verbosity': 2})
```

215 Because the `smuggle()` function is defined in the notebook namespace, it is  
216 also possible (though never necessary) to call it directly. Deactivating `davos`  
217 will delete the name “`smuggle`” from the namespace, unless its value has  
218 been overwritten and no longer refers to the `smuggle()` function. It will also  
219 deregister the `davos` parser from the set of input transformers run when each  
220 notebook cell is executed. While the overhead added by the `davos` parser is  
221 de minimis, this may be useful, for example, when optimizing or precisely  
222 profiling code.

### 223 3. Illustrative Examples

### 224 4. Impact

225 Like virtual environments, containers, and virtual machines, the `davos` li-  
226 brary (when used in conjunction with Jupyter notebooks) provides a lightweight  
227 mechanism for sharing code and ensuring reproducibility across users and  
228 computing environments (Fig. 1). Further, `davos` enables users to fully  
229 specify (and install, as needed) any project dependencies within the same

230 notebook. This provides a system whereby executable code (along with text  
231 and media) *and* code for setting up and configuring the project dependencies,  
232 may be combined within a single notebook file.

233 We designed **davos** for use in research applications. For example, in many  
234 settings **davos** may be used as a drop-in replacement for more-difficult-to-  
235 set-up virtual environments, containers, and/or virtual machines. For re-  
236 searchers, this lowers barriers to sharing code. By eliminating most of the  
237 setup costs of reconstructing the original researchers’ computing environ-  
238 ment, **davos** also lowers barriers to entry for members of the scientific com-  
239 munity and the public who seek to *benefit* from shared code.

240 Beyond research applications, **davos** is also useful in pedagogical settings.  
241 For example, in programming courses, instructors and students may import  
242 the **davos** library into their notebooks to provide a simple means of ensur-  
243 ing their code will run on others’ machines. When combined with online  
244 notebook-based platforms like Google Colaboratory, **davos** provides a con-  
245 venient way to manage dependencies within a notebook, without requiring  
246 any software (beyond a web browser) to be installed on the students’ or in-  
247 structors’ systems. For the same reasons, **davos** also provides an elegant  
248 means of sharing ready-to-run notebook-based demonstrations that install  
249 their dependencies automatically.

250 Since its initial release, **davos** has found use in a variety of applications.  
251 In addition to managing computing environments for multiple ongoing re-  
252 search studies, **davos** is being used by both students and instructors in pro-  
253 gramming and methods courses such as Storytelling with Data [18] (an open  
254 course on data science, visualization, and communication) and Laboratory  
255 in Psychological Science [19] (an open course on experimental and statistical  
256 methods for psychology research) to simplify distributing lessons and sub-  
257 mitting assignments, as well as in online demos such as **abstract2paper** [20]  
258 (an example application of GPT-Neo [21, 22]) to share ready-to-run code  
259 that installs dependencies automatically.

260 Our work also has several more subtle “advanced” use cases and poten-  
261 tial impacts. Whereas Python’s built-in **import** statement is agnostic to  
262 packages’ version numbers, **smuggle** statements (when combined with onion  
263 comments) are version-sensitive. And because onion comments are parsed  
264 at runtime, required package and their specified versions are installed in a  
265 just-in-time manner. Thus, it is possible in most cases to **smuggle** a specific  
266 package version or revision even if a different version has already been loaded.  
267 This enables more complex uses that take advantage of multiple versions of  
268 a package within a single interpreter session. This could be useful in cases  
269 where specific features are added or removed from a package across differ-  
270 ent versions, or in comparing the performance or functionality of particular

271 features across different versions of the same package.

272 A second advanced use case is in providing a proof-of-concept of how one  
273 can add new “keyword-like” operators to the Python language by leverag-  
274 ing notebooks’ error-handling mechanisms. This could lead to exciting new  
275 tools that, like `davos`, extend the Python language in useful ways within  
276 notebook-based environments. We note that our approach to adding the  
277 `smuggle` keyword to Python when `davos` is imported into a notebook-based  
278 environment also has the potential to be exploited for more nefarious pur-  
279 poses. For example, a malicious user could use a similar approach (e.g.,  
280 in a different library) to substantially change a notebook’s functionality by  
281 adding new *unexpected* keyword-like objects (e.g., based around common ty-  
282 pos). This could lead to difficult-to-predict changes in a notebook’s behavior  
283 once the malicious library was imported. This highlights an important rea-  
284 son why security-conscious users would be well-served to only make use of  
285 libraries from trusted sources, or whose code is publicly available for review.

## 286 5. Conclusions

287 The `davos` library supports reproducible research by providing a novel  
288 lightweight system for sharing notebook-based code. But perhaps the most  
289 exciting uses of the `davos` library are those that we have *not* yet considered  
290 or imagined. We hope that the Python community will find `davos` to pro-  
291 vide a convenient means of managing project dependencies to facilitate code  
292 sharing. We also hope that some of the more advanced applications of our  
293 library might lead to new insights or discoveries.

## 294 Author Contributions

295 **Paxton C. Fitzpatrick:** Conceptualization, Methodology, Software,  
296 Validation, Writing - Original Draft, Visualization. **Jeremy R. Manning:**  
297 Conceptualization, Resources, Validation, Writing - Review & Editing, Su-  
298 pervision, Funding acquisition.

## 299 Funding

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

## 303 Declaration of Competing Interest

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

## 307 Acknowledgements

308 We acknowledge useful feedback and discussion from the students of  
309 JRM's *Storytelling with Data* course (Winter, 2022 offering) who used pre-  
310 liminary versions of our library in several assignments.

## 311 References

- 312 [1] G. van Rossum, Python reference manual, Department of Computer  
313 Science [CS] (R 9525) (1995).
- 314 [2] Python Software Foundation, The Python Package Index (PyPI),  
315 <https://pypi.org> (2003).
- 316 [3] conda-forge community, The conda-forge Project: Community-based  
317 Software Distribution Built on the conda Package Format and Ecosys-  
318 tem, <https://doi.org/10.5281/zenodo.4774217> (July 2015). doi:  
319 [10.5281/zenodo.4774217](https://doi.org/10.5281/zenodo.4774217).
- 320 [4] N. Coghlan, D. Stuft, Version Identification and Dependency Specifica-  
321 tion, PEP 440, Python Software Foundation (March 2013).
- 322 [5] B. Cannon, N. Smith, D. Stuft, Specifying minimum build system re-  
323 quirements for python projects, PEP 518, Python Software Foundation  
324 (May 2016).
- 325 [6] Anaconda, Inc., conda, <https://docs.conda.io> (2012).
- 326 [7] S. Eustace, Poetry: Python packaging and dependency management  
327 made easy, <https://github.com/python-poetry/poetry> (December  
328 2019).
- 329 [8] T. Kluyver, B. Ragan-Kelley, F. Pérez, B. Granger, M. Bussonnier,  
330 J. Frederic, K. Kelley, J. Hamrick, J. Grout, S. Corlay, P. Ivanov,  
331 D. Avila, S. Abdalla, C. Willing, Jupyter Notebooks – a publish-  
332 ing format for reproducible computational workflows, in: F. Loizides,  
333 B. Schmidt (Eds.), Positioning and Power in Academic Publishing: Play-  
334 ers, Agents and Agendas, IOS Press, Netherlands, 2016, pp. 87–90.  
335 doi:[10.3233/978-1-61499-649-1-87](https://doi.org/10.3233/978-1-61499-649-1-87).

- 336 [9] R. P. Goldberg, Survey of virtual machine research, *Computer* 7 (6)  
337 (1974) 34–45.
- 338 [10] Y. Altintas, C. Brecher, M. Weck, S. Witt, Virtual Machine Tool,  
339 *CIRP Annals* 54 (2) (2005) 115–138. doi:[https://doi.org/10.1016/](https://doi.org/10.1016/S0007-8506(07)60022-5)  
340 [S0007-8506\(07\)60022-5](https://doi.org/10.1016/S0007-8506(07)60022-5).
- 341 [11] M. Rosenblum, VMware’s Virtual Platform: A virtual machine monitor  
342 for commodity PCs, in: *IEEE Hot Chips Symposium*, IEEE, 1999, pp.  
343 185–196.
- 344 [12] D. Merkel, Docker: lightweight linux containers for consistent develop-  
345 ment and deployment, *Linux Journal* 239 (2) (2014) 2.
- 346 [13] G. M. Kurtzer, V. Sochat, M. W. Bauer, Singularity: Scientific contain-  
347 ers for mobility of compute, *PLoS One* 12 (5) (2017) e0177459.
- 348 [14] F. Pérez, B. E. Granger, IPython: a system for interactive scientific  
349 computing, *Computing in science and engineering* 9 (3) (2007) 21–29.  
350 doi:[10.1109/MCSE.2007.53](https://doi.org/10.1109/MCSE.2007.53).
- 351 [15] G. van Rossum, J. Lehtosalo, L. Langa, Type Hints, PEP 484, Python  
352 Software Foundation (September 2014).
- 353 [16] L. Torvalds, J. Hamano, Git: Fast version control system, [https://](https://git.kernel.org/pub/scm/git/git.git)  
354 [git.kernel.org/pub/scm/git/git.git](https://git.kernel.org/pub/scm/git/git.git) (April 2005).
- 355 [17] K. Thompson, Programming Techniques: Regular expression search al-  
356 gorithm, *Communications of the ACM* 11 (6) (1968) 419–422. doi:  
357 [10.1145/363347.363387](https://doi.org/10.1145/363347.363387).
- 358 [18] J. R. Manning, *Storytelling with Data*, [https://github.com/](https://github.com/ContextLab/storytelling-with-data)  
359 [ContextLab/storytelling-with-data](https://github.com/ContextLab/storytelling-with-data) (June 2021). doi:[10.5281/](https://doi.org/10.5281/zenodo.5182775)  
360 [zenodo.5182775](https://doi.org/10.5281/zenodo.5182775).
- 361 [19] J. Manning, *ContextLab/experimental-psychology: v1.0* (Spring, 2022),  
362 [https://github.com/ContextLab/experimental-psychology/tree/](https://github.com/ContextLab/experimental-psychology/tree/v1.0)  
363 [v1.0](https://github.com/ContextLab/experimental-psychology/tree/v1.0) (May 2022). doi:[10.5281/zenodo.6596762](https://doi.org/10.5281/zenodo.6596762).
- 364 [20] J. R. Manning, *abstract2paper*, [https://github.com/ContextLab/](https://github.com/ContextLab/abstract2paper)  
365 [abstract2paper](https://github.com/ContextLab/abstract2paper) (June 2021).
- 366 [21] L. Gao, S. Biderman, S. Black, L. Golding, T. Hoppe, C. Foster,  
367 J. Phang, H. He, A. Thite, N. Nabeshima, S. Presser, C. Leahy, The  
368 Pile: An 800GB Dataset of Diverse Text for Language Modeling, *arXiv*  
369 preprint arXiv:2101.00027 (2020).

370 [22] S. Black, L. Gao, P. Wang, C. Leahy, S. Biderman, GPT-Neo: Large  
371 Scale Autoregressive Language Modeling with Mesh-Tensorflow, [http:](http://github.com/eleutherai/gpt-neo)  
372 [//github.com/eleutherai/gpt-neo](http://github.com/eleutherai/gpt-neo) (2021).