

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 that must be packaged alongside the primary code base. Here we propose a lighter-weight solution than virtual environments: the **davos** library. When used in combination with a notebook-based Python project, the **davos** library provides a mechanism for specifying (and automatically installing) the correct package versions of the project’s dependencies. The **davos** library also ensures that those versions are in use any 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	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, and 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	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

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 in and of itself is often insufficient for enabling researchers to reproduce each others' work.

Within the Python [1] community, external packages that are published in the most popular repositories [2, 3] are associated with version numbers and tags that enable users to guarantee that they are installing exactly the same

code across different computing environments. Despite that it is *possible* to manually install the intended version numbers of every dependency of a Python script or package, manually tracking down those dependencies can impose a substantial burden on the user.

Researchers, programmers, and others have developed a broad set of approaches and tools to facilitate code sharing and reproducible outcomes (Fig. 1). At one extreme, simply publishing a set of Python scripts (.py files) may enable others to use or gain insights into the relevant work. Because Python is installed by default on most modern operating systems, for some projects this may be sufficient. Another popular approach entails creating JSON files, called Jupyter notebooks [4], that comprise a mix of text, executable code, and embedded media. Notebooks may call or import external scripts or libraries in order to provide a more compact and readable experience for users. Each of these systems (Python scripts and notebooks) provides a convenient means of sharing code, with the caveat that they do not specify the computing environment in which the code is executed. Therefore the functionality of code shared using these systems cannot be guaranteed across different computing environments.

At another extreme, virtual machines [5, 6, 7] provide a hardware-level simulation of the desired system. Virtual machines are typically isolated from the user’s system, such that installing or running software on a virtual machine does not impact the user’s primary operating system or computing environment. Containers [e.g., 8, 9] provide a similar “isolated” experience. Although containerized environments do not specify hardware-level operations, they are typically packaged with a complete operating system, in addition to a complete copy of Python and any relevant package dependencies. Virtual environments [e.g., 10] also provide a computing environment that is largely separated from the user’s main environment. They incorporate a copy of Python and the target software’s dependencies, but virtual environments do not specify or reproduce an operating system for the runtime environment. Each of these systems (virtual machines, containers, and virtual environments) guarantees (to differing degrees— at the hardware level, operating system level, and Python environment level, respectively) that the relevant code will run similarly for different users. However, each of these systems also relies on additional software that can be resource intensive or burdensome to install or configure.

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 allows researchers to include text, executable code, and media within a single file. No further setup or installation is required, beyond what

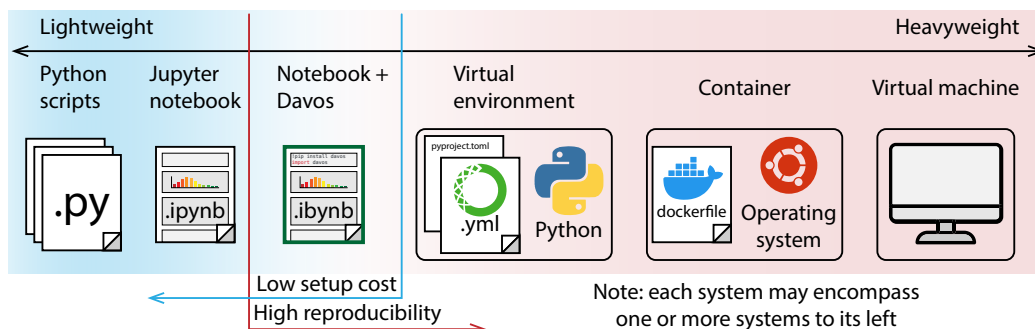


Figure 1: **Systems for sharing code within the Python ecosystem.** From left to right: plain-text **Python scripts** (`.py` files) provide the most basic “system” for sharing raw code. Scripts may reference external libraries, but those libraries must be manually installed on other users’ systems. Further, 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 the required external libraries (at their correct versions, when specified). **Virtual environments** install an isolated copy of Python and all required dependencies. This typically requires defining a `requirements.txt` file or an environment (`.yml`) file that specifies all project dependencies (including version numbers of external libraries). **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 virtual 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.

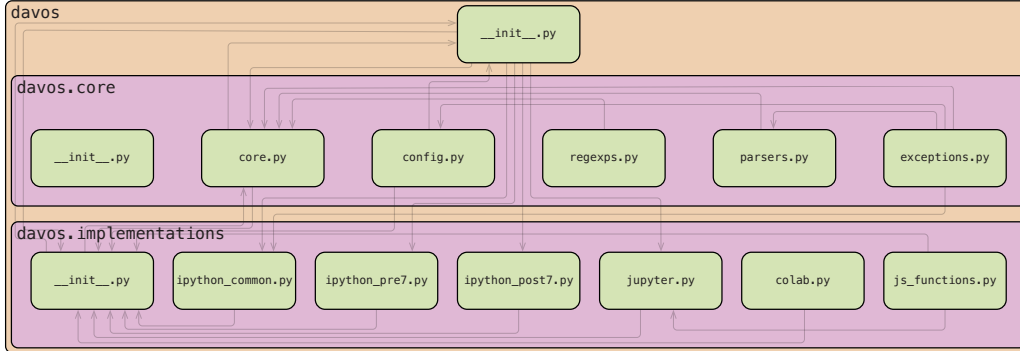


Figure 2: **Package structure.**

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

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 implementation approaches depending on various 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 [11] 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

79 variants simply by creating a new `davos.implementations` module with any
80 necessary tweaks to existing helper functions.

81 *2.2. Software functionalities*

82 *2.2.1. The `smuggle` statement*

83 Importing `davos` in an IPython notebook enables an additional Python
84 keyword: “`smuggle`” (see Section 2.3 for details on how this works). The
85 `smuggle` statement can be used as a drop-in replacement for Python’s built-
86 in `import` statement to load libraries, modules, and other objects into the
87 current namespace. However, whereas `import` will fail if the requested pack-
88 age is not installed locally, `smuggle` statements can handle missing packages
89 on the fly. If a smuggled package does not exist in the local environment,
90 `davos` will install it automatically, expose its contents to Python’s `import`
91 machinery, and load it into the namespace for immediate use.

92 *2.2.2. The onion comment*

93 For greater control over the behavior of `smuggle` statements, `davos` de-
94 fines an additional construct called the “onion comment”. An onion comment
95 is a special type of inline comment that may be placed on a line containing a
96 `smuggle` statement to customize how `davos` searches for the smuggled pack-
97 age locally and, if necessary, downloads and installs it. Onion comments
98 follow a simple syntax based on the “type comment” syntax introduced in
99 PEP 484 [12], and are designed to make managing packages with `davos` intu-
100 itive and familiar. To construct an onion comment, simply provide the name
101 of the installer program (e.g., `pip`) and the same arguments one would use
102 to manually install the package as desired via the command line:

```
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
```

103

104 Onion comments are useful when smuggling a package whose distribution
105 name (i.e., the name used when installing it) is different from its top-level
106 module name (i.e., the name used when importing it):

```

smuggle dateutil      # pip: python-dateutil
from sklearn.decomposition smuggle PCA      # pip: scikit-learn

```

107

108 However, the most powerful use of the onion comment is making **smuggle**
 109 statements *version-sensitive*. If an onion comment includes a version spec-
 110 ifier [13], **davos** will ensure that the version of the package loaded into the
 111 notebook always matches the specific version requested, or satisfies the given
 112 version constraints. If the smuggled package exists locally, **davos** will extract
 113 its version info from its metadata and compare it to the specifier provided. If
 114 the two are incompatible (or no local installation is found), **davos** will install
 115 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

```

116

117 Onion comments can similarly be used to smuggle specific VCS references
 118 (e.g., Git [14] 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

```

119

120 **davos** processes onion comments internally before forwarding arguments to
 121 the installer program. In addition to preventing onion comments from being
 122 used as a vehicle for shell injection attacks, this allows **davos** take certain logi-
 123 cal actions when particular arguments are passed. For example, the **--force-**
 124 **reinstall**, **-I/--ignore-installed**, and **-U/--upgrade** flags will all cause
 125 **davos** to skip searching for a smuggled package locally before installing a new
 126 copy; **--no-input** will temporarily enable **davos**'s non-interactive mode (see
 127 Section 2.2.2); and installing a package into **<dir>** with **--target <dir>**
 128 will cause **dir** to be prepended to the module search path (**sys.path**), if
 129 necessary, so the package can be imported:

```

# 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

```

130

131 *2.2.3. The `davos` config*

132 The `davos` config object provides a simple, high-level interface that allows
133 users to view and set various options that affect `davos`'s behavior. After
134 importing `davos`, the config instance (a singleton) for the current session is
135 available as `davos.config`, and its various fields are accessible as attributes.
136 The config object exposes a mixture of writable and read-only fields. Writable
137 fields include:

- 138 • `.active`: Whether or not `davos` functionality (i.e., support for `smug-`
139 `gle` statements and onion comments) should be enabled for subsequent
140 code. Defaults to `True` when `davos` is first imported. See Section 2.3
141 for additional info.
- 142 • `.auto_rerun`: Controls behavior if `davos` is used to `smuggle` a new ver-
143 sion of a package that was previously imported and cannot be reloaded
144 (i.e., it contains C-extensions that dynamically generate code). If `True`
145 (default: `False`), `davos` will automatically restart the notebook kernel
146 and rerun all code up to (and including) the current `smuggle` state-
147 ment. Otherwise, `davos` will issue a warning, pause execution, and
148 prompt the user with buttons to either restart & rerun the notebook
149 or continue running with the imported package version. (Note: not
150 configurable in Google Colaboratory).
- 151 • `.confirm_install`: If `True` (default: `False`), `davos` will require user
152 confirmation (`[y]es/[n]o` input) before installing a smuggled package.
- 153 • `.noninteractive`: Setting to `True` (default: `False`) enables non-interactive
154 mode, in which all user input and confirmation is disabled. Note that
155 in non-interactive mode, the `confirm_install` option is set to `False`,
156 and if `auto_rerun` is `False`, `davos` will throw an error if a smuggled
157 package cannot be reloaded.
- 158 • `.pip_executable`: The path to the `pip` executable used to install
159 smuggled packages. Default is programmatically determined from the
160 Python environment and falls back to `sys.executable -m pip` if one
161 can't be found.
- 162 • `.suppress_stdout`: If `True` (default: `False`), suppress all unnecessary
163 output issued by both `davos` and the installer program. Useful when
164 smuggling packages that need to install many dependencies and/or gen-
165 erate extensive output. If the installer program throws an error, both
166 stdout and stderr will be shown with the traceback.

167 The top-level `davos` namespace additionally defines a handful of convenience
168 functions for setting and checking `davos`’s active/inactive state (`davos.activate()`;
169 `davos.deactivate()`; `davos.is_active()`) as well as the `davos.configure()`
170 function, which allows setting multiple config fields at once.

171 2.2.4. *Advanced functionality*

172 Because `davos` parses onion comments at runtime, required packages are
173 validated and installed in a just-in-time manner. Thus, it is possible in most
174 cases to `smuggle` a specific package version or revision

175 2.3. *Implementation details*

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

185 The `smuggle` keyword can be enabled and disabled at any time by “ac-
186 tivating” and “deactivating” `davos` (see Section 2.2.3, above). When `davos`
187 is first imported, it is activated automatically. Activating `davos` triggers
188 two actions: (1) the `smuggle()` function is injected into the IPython user
189 namespace, and (2) the `davos` parser is registered as a custom IPython input
190 transformer. IPython preprocesses all executed code as plain text before it is
191 sent to the Python parser, in order to handle special constructs like `%magic`
192 and `!shell` commands. `davos` hooks into this process to transform `smuggle`
193 statements into syntactically valid Python code. The `davos` parser uses a
194 complex regular expression [15] to match lines of code containing `smuggle`
195 statements (and, optionally, onion comments), extract relevant information
196 from their text, and replace them with equivalent calls to the `smuggle()`
197 function. For example, if a user runs a notebook cell containing

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

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

```
200 smuggle(name="numpy", as_="np", installer="pip", args_str="numpy>1.16,<=1.20 -  
201 vv", installer_kwargs={'editable': False, 'spec': 'numpy>1.16,<=1.20', 'verbos
```

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

210 3. Illustrative Examples

211 4. Impact

212 Like virtual environments, containers, and virtual machines, the `davos` li-
213 brary (when used in conjunction with Jupyter notebooks) provides a lightweight
214 mechanism for sharing code and ensuring reproducibility across users and
215 computing environments (Fig. 1). Further, `davos` enables users to fully
216 specify (and install, as needed) any project dependencies within the same
217 notebook. This provides a system whereby executable code (along with text
218 and media) *and* code for setting up and configuring the project dependencies,
219 may be combined within a single notebook file.

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

227 Beyond research applications, `davos` is also useful in pedagogical settings.
228 For example, in programming courses, instructors and students may import
229 the `davos` library into their notebooks to provide a simple means of ensur-
230 ing their code will run on others’ machines. When combined with online
231 notebook-based platforms like Google Colaboratory, `davos` provides a con-
232 venient way to manage dependencies within a notebook, without requiring
233 any software (beyond a web browser) to be installed on the students’ or in-
234 structors’ systems. For the same reasons, `davos` also provides an elegant
235 means of sharing ready-to-run notebook-based demonstrations that install
236 their dependencies automatically.

237 Since its initial release, `davos` has found use in a variety of applications. In
238 addition to managing computing environments for multiple ongoing research
239 studies, `davos` is being used by both students and instructors in programming

240 courses such as *Storytelling with Data* [16] (an open course on data science,
241 visualization, and communication) to simplify distributing lessons and sub-
242 mitting assignments, as well as in online demos such as `abstract2paper`
243 [17] (an example application of GPT-Neo) to share ready-to-run code that
244 installs dependencies automatically.

245 Our work also has several more subtle “advanced” use cases and poten-
246 tial impacts. Whereas Python’s built-in `import` statement is agnostic to
247 packages’ version numbers, `smuggle` statements (when combined with onion
248 comments) are version-sensitive. This enables multiple versions of a single li-
249 brary to be imported within the same notebook. This could be useful in cases
250 where specific features were added or removed from a package across differ-
251 ent versions, or in comparing the performance or functionality of particular
252 features across different versions of the same package.

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

267 5. Conclusions

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

275 Author Contributions

276 **Paxton C. Fitzpatrick:** Conceptualization, Methodology, Software,
277 Validation, Writing - Original Draft, Visualization. **Jeremy R. Manning:**

278 Conceptualization, Resources, Validation, Writing - Review & Editing, Su-
279 pervision, Funding acquisition.

280 **Funding**

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

284 **Declaration of Competing Interest**

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

288 **Acknowledgements**

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

292 **References**

- 293 [1] G. van Rossum, Python reference manual, Department of Computer
294 Science [CS] (R 9525) (1995).
- 295 [2] Python Software Foundation, The Python Package Index (PyPI),
296 <https://pypi.org> (2003).
- 297 [3] conda-forge community, The conda-forge Project: Community-based
298 Software Distribution Built on the conda Package Format and Ecosys-
299 tem, <https://doi.org/10.5281/zenodo.4774217> (July 2015). doi:
300 [10.5281/zenodo.4774217](https://doi.org/10.5281/zenodo.4774217).
- 301 [4] T. Kluyver, B. Ragan-Kelley, F. Pérez, B. Granger, M. Bussonnier,
302 J. Frederic, K. Kelley, J. Hamrick, J. Grout, S. Corlay, P. Ivanov,
303 D. Avila, S. Abdalla, C. Willing, Jupyter notebooks – a publish-
304 ing format for reproducible computational workflows., in: F. Loizides,
305 B. Schmidt (Eds.), Positioning and Power in Academic Publishing: Play-
306 ers, Agents and Agendas, IOS Press, Netherlands, 2016, pp. 97–90.
307 doi:[10.3233/978-1-61499-649-1-87](https://doi.org/10.3233/978-1-61499-649-1-87).

- [5] R. P. Goldberg, Survey of virtual machine research, *Computer* 7 (6) (1974) 34–45.
- [6] Y. Altintas, C. Brecher, M. Weck, S. Witt, *Virtual machine tool*, *CIRP Annals* 54 (2) (2005) 115–138. doi:[https://doi.org/10.1016/S0007-8506\(07\)60022-5](https://doi.org/10.1016/S0007-8506(07)60022-5).
URL <https://www.sciencedirect.com/science/article/pii/S0007850607600225>
- [7] M. Rosenblum, VMware’s Virtual Platform: A virtual machine monitor for commodity PCs, in: *IEEE Hot Chips Symposium*, IEEE, 1999, pp. 185–196.
- [8] D. Merkel, Docker: lightweight linux containers for consistent development and deployment, *Linux Journal* 239 (2) (2014) 2.
- [9] G. M. Kurtzer, V. Sochat, M. W. Bauer, Singularity: Scientific containers for mobility of compute, *PLoS One* 12 (5) (2017) e0177459.
- [10] Anaconda, Inc., conda, <https://docs.conda.io> (2012).
- [11] 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](https://doi.org/10.1109/MCSE.2007.53).
- [12] G. van Rossum, J. Lehtosalo, L. Langa, *Type Hints*, PEP 484, Python Software Foundation (September 2014).
URL <https://www.python.org/dev/peps/pep-0484>
- [13] N. Coghlan, D. Stufft, *Version identification and dependency specification*, PEP 440, Python Software Foundation (March 2013).
URL <https://peps.python.org/pep-0440>
- [14] L. Torvalds, J. Hamano, Git: Fast version control system, <https://git.kernel.org/pub/scm/git/git.git> (April 2005).
- [15] K. Thompson, Programming Techniques: Regular expression search algorithm, *Communications of the ACM* 11 (6) (1968) 419–422. doi:[10.1145/363347.363387](https://doi.org/10.1145/363347.363387).
- [16] J. R. Manning, Data wrangler, Zenodo 10.5281/zenodo.5123310 (2021).
- [17] J. R. Manning, Episodic memory: mental time travel or a quantum “memory wave” function?, *Psychological Review* 128 (4) (2021) 711–725.