

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 | 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 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

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

Computational researchers and other programmers have developed a broad set of approaches and tools to facilitate code sharing and reproducible outcomes (Fig. 1). At one extreme, simply distributing 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 Jupyter notebooks [8] that comprise a mix of text, executable code, and embedded media. Notebooks may call or import external scripts or libraries—even intersperse snippets of other programming or markup languages—in order to provide a more compact and readable experience for users. Both of these systems (Python scripts and notebooks) provide 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 users or setups.

At another extreme, virtual machines [9, 10, 11] provide a hardware-level simulation of the desired system. Virtual machines are typically isolated 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., 12, 13] 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., 6, 7] 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 complex or resource-intensive to install and use, creating potential barriers to both contributing to and taking advantage of open science resources.

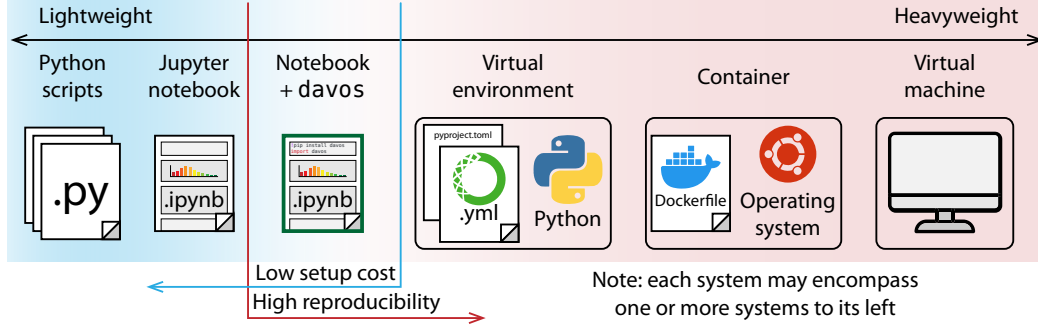


Figure 1: **Systems for sharing code within the Python ecosystem.** From left to right: plain-text **Python scripts** (.py files) provide the most basic “system” for sharing raw code. Scripts may reference external libraries, but those libraries must be manually installed on other users’ systems. Further, 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.

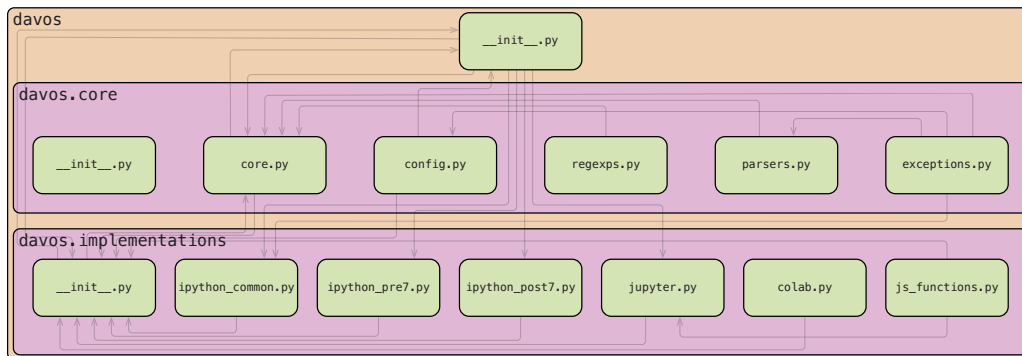


Figure 2: **Package structure.**

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**

78 subpackage. This second subpackage defines multiple, interchangeable ver-
79 sions of each helper function, organized into modules by the conditions that
80 trigger their use. At runtime, **davos** detects various features in the notebook
81 environment and selectively imports a single version of each helper function
82 into the top-level **davos.implementations** namespace, allowing **davos.core**
83 modules to access the correct implementations for the current notebook en-
84 vironment in a single, consistent location. An additional benefit of this de-
85 sign pattern is that it allows maintainers or users to easily extend **davos**
86 to support new, updated, or custom notebook variants by creating a new
87 **davos.implementations** module with any necessary tweaks to the existing
88 helper functions.

89 *2.2. Software functionalities*

90 *2.2.1. The **smuggle** statement*

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

100 *2.2.2. The onion comment*

101 For greater control over the behavior of **smuggle** statements, **davos** de-
102 fines an additional construct called the “onion comment.” An onion comment
103 is a special type of inline comment that may be placed on a line containing a
104 **smuggle** statement to customize how **davos** searches for the smuggled pack-
105 age locally and, if necessary, downloads and installs it. Onion comments
106 follow a simple format based on the “type comment” syntax introduced in
107 PEP 484 [15], and are designed to make managing packages with **davos** intu-
108 itive and familiar. To construct an onion comment, users provide the name
109 of the installer program (e.g., **pip**) and the same arguments one would use
110 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 prior to 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
```

138 Similarly, passing `--no-input` will temporarily enable **davos**'s non-interactive
139 mode (see Section 2.2.3), 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. 4 for example usage). These include:

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

164 next time the kernel is restarted manually. (Note: not configurable in
165 Google Colaboratory).

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

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

188 2.3. Implementation details

189 Functionally, importing `davos` appears to make “`smuggle`” a valid Python
190 keyword, similar to standard keywords like “`import`”, “`def`”, or “`return`”.
191 It also appears to fundamentally change how Python treats comments, sud-
192 denly allowing them to potentially influence code behavior at runtime if they
193 conform to a particular syntax. However, `davos` doesn't actually modify the
194 rules of Python's parser or lexical analyzer in order to accomplish this—
195 in fact, modifying the Python grammar isn't possible at runtime, as doing
196 so would require rebuilding the interpreter. Instead, `davos` leverages the
197 IPython notebook backend to implement the `smuggle` statement and onion
198 comment via a combination of namespace injections and its own (far simpler)
199 custom parser.

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

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

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

```
215 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})
```

216 The call to the `smuggle()` function then carries out `davos`’s central logic
217 of determining whether the smuggled package should be installed, doing so
218 if necessary, and subsequently loading it into the namespace. This process
219 is outlined in Figure 3. Because the `smuggle()` function is defined in the
220 notebook namespace, it is also possible (though never necessary) to call
221 it directly. Deactivating `davos` will delete the name “`smuggle`” from the
222 namespace, unless its value has been overwritten and no longer refers to the
223 `smuggle()` function. It will also deregister the `davos` parser from the set of
224 input transformers run when each notebook cell is executed. While the over-
225 head added by the `davos` parser is minimal, this may be useful, for example,
226 when optimizing or precisely profiling code.

227 3. Illustrative Examples

228 Figure 4 illustrates how one might use `davos` in a real-world setting to
229 facilitate reproducible data analyses. In this scenario, a user has acquired a
230 dataset and pre-trained semantic model (possibly shared by a collaborator,
231 or downloaded from a public repository) with which they want to run a set

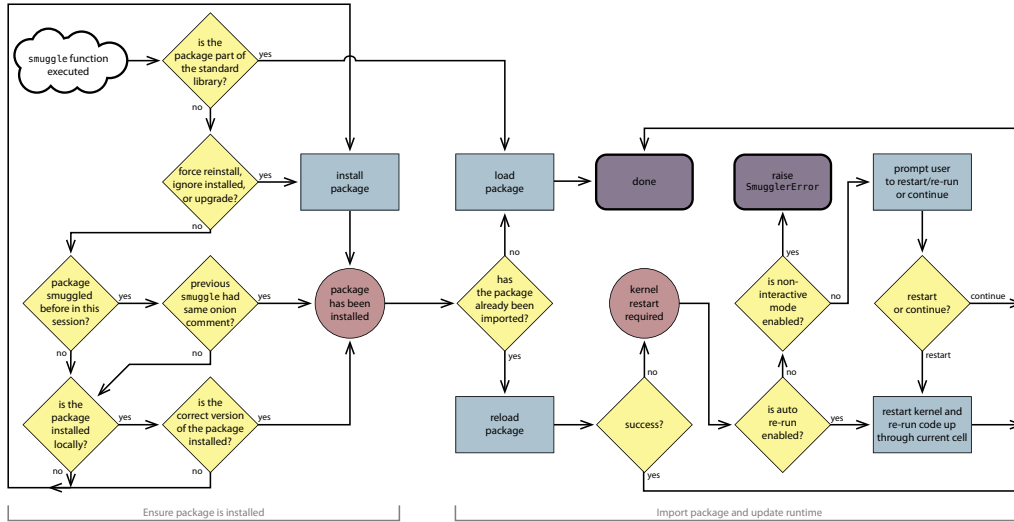


Figure 3: `smuggle()` function logic.

of analyses in a Jupyter notebook. The hypothetical dataset (a `pandas` [18] Panel) and model (a `scikit-learn` [19] Pipeline) were created and saved using the latest versions of each package available in June 2019 and, due to their age and the formats in which they were provided, now pose certain challenges to loading and using them. The code in Figure 4 would be included at the top of the user’s analysis notebook, and demonstrates how `davos` enables them to overcome these issues cleanly and efficiently, while simultaneously constructing a reproducible Python environment that ensures their analyses are always run under the same conditions, both by the user themselves and anyone with whom they share their code.

After importing `davos` (line 1), the user first smuggles two utilities for interacting with local files in the code below. The `smuggle` statement in line 3 loads the `is_file()` function from the Python standard library’s `os.path` module. Standard library modules are included with all Python distributions, so this line is functionally equivalent to an `import` statement and no onion comment is needed. Line 4 loads the `joblib` library [20], installing it first, if necessary. Since `joblib`’s I/O interface has historically remained stable and backwards-compatible across releases, requiring that users have a particular exact version installed would likely be unnecessarily restrictive. However, there’s also no guarantee that a *future* release won’t introduce some breaking change, so to help ensure the analysis notebook continues to run properly in the future, the onion comment in line 4 limits allowable versions to those already released when the code was written.

Line 6 then uses the `davos.config` object to enable `davos`’s `auto_re-`

```

1  import davos
2
3  from os.path smuggle is_file
4  smuggle joblib                                # pip: joblib<=1.2.0
5
6  davos.config.auto_rerun = True
7  smuggle numpy as np                          # pip: numpy==1.21.6
8
9  if not is_file("~/datasets/data-new.csv"):
10     smuggle pandas as pd                      # pip: pandas<0.25.0
11     tmp_data = pd.read_pickle("~/datasets/data-old.p")
12     tmp_data.to_frame().to_csv("~/datasets/data-new.csv")
13
14  smuggle pandas as pd                        # pip: pandas==1.3.5
15
16  davos.configure(auto_rerun=False, suppress_stdout=True, noninteractive=True)
17  smuggle tensorflow as tf                    # pip: tensorflow==2.9.2
18  smuggle hypertools as hyp                  # pip: hypertools==0.8.0
19  davos.configure(suppress_stdout=False, noninteractive=False)
20
21  smuggle matplotlib.pyplot as plt           # pip: matplotlib==3.5.3
22  smuggle seaborn as sns                     # pip: seaborn==0.12.1
23  smuggle quail                              # pip: git+https://github.com/myfork/quail@6c847a4
24
25  davos.config.pip_executable = "~/envs/nb-server/bin/pip"
26  smuggle widgetsnbextension as _           # pip: widgetsnbextension==3.5.2
27  davos.config.pip_executable = "~/envs/nb-kernel/bin/pip"
28  smuggle ipywidgets                         # pip: ipywidgets==7.6.5
29
30  from tqdm.notebook smuggle tqdm            # pip: tqdm==4.62.3
31
32  data = pd.read_csv("~/datasets/data-new.csv", index_col=[0, 1])
33  smuggle sklearn                            # pip: scikit-learn<0.22.0 --no-use-pep517
34  transformer = joblib.load("~/models/text-transformer.joblib")
35  smuggle sklearn                            # pip: scikit-learn==1.1.3

```

Figure 4: Example use case for davos's functionality.

run option before smuggling the next two packages, NumPy [21] and pandas. Because these packages rely heavily on custom C data types, loading them may require restarting the notebook kernel if different versions were already imported during the same interpreter session (see Section 2.2.3). This is unlikely to happen with NumPy, as it is loaded explicitly only once (by line 7) and is not required by any packages loaded before it (and therefore not implicitly imported while loading them). However, it's possible a future user could edit this notebook to add additional code before line 7, run the notebook's cells out of order, or even configure their IPython kernel to execute custom code on startup (e.g., as Google Colab notebooks do), any of which could potentially import a different, existing version of NumPy before `davos` smuggles the version specified in the onion comment. Enabling the `auto_re-run` option before smuggling NumPy helps to simplify running the notebook in these scenarios, but the main benefit it affords comes when smuggling the `pandas` library.

The dataset that the user wants to analyze was provided as a `pandas.Panel` object, serialized with Python's built-in `pickle` module. The `pickle` protocol is a popular choice for data persistence in Python, allowing users to save, share, and load arbitrary objects, though with an important caveat: A "pickled" object's class definition must be importable from the same module as when it was saved in order to successfully "unpickle" (i.e., load and restore) it later. The `Panel` class was removed from `pandas` in July, 2019 (v0.25.0), meaning that the user's dataset can be loaded only if their notebook uses a `pandas` version released prior to that change. However, updates to the `pandas` library since then have brought substantial improvements that the user would like to take advantage of in their analysis code, including faster performance, more memory-efficient data types, new functions and methods, compatibility with newer versions of Python and popular plotting libraries, and better display formats (including in Jupyter notebooks). `davos` makes it possible to do both within a single notebook, and with the `auto_rerun` option, this process can be fully automated. The first time the notebook is run, lines 10–12 will install and load an older `pandas` version that defines the `Panel` class, use it to read the dataset from its serialized file format, convert the dataset from a `Panel` to a multi-index `DataFrame`, and write its contents to a CSV file. The `smuggle` statement on line 14 then installs a newer version of `pandas` (v1.3.5) for use in the analyses themselves. Because an older `pandas` version was previously loaded (by line 10), and because the package's C extension modules were modified between the old and new versions, switching to the new version requires restarting the notebook kernel. With `auto_rerun` enabled, `davos` takes care of this automatically and will restart the kernel, re-run the notebook up through

the current cell, and also re-queue any other cells that the user had queued before the restart was triggered. This second time the code is run (as well as any future times), `smuggle` statements that have already been executed once before will run (virtually) as quickly as import statements, since the specified package versions are already installed. Additionally,

4. Impact

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

We designed `davos` for use in research applications. For example, in many settings, `davos` may be used as a drop-in replacement for more-difficult-to-set-up virtual environments, containers, and/or virtual machines. For researchers, this lowers barriers to both sharing code. By eliminating most of the setup costs of reconstructing the original researchers' computing environment, `davos` also lowers barriers to entry for members of the scientific community and the public who seek to benefit from shared code.

Beyond research applications, `davos` is also useful in pedagogical settings. For example, in programming courses, instructors and students may import the `davos` library into their notebooks to provide a simple means of ensuring their code will run on others' machines. When combined with online notebook-based platforms like Google Colaboratory, `davos` provides a convenient way to manage dependencies within a notebook, without requiring any software (beyond a web browser) to be installed on the students' or instructors' systems. For the same reasons, `davos` also provides an elegant means of sharing ready-to-run notebook-based demonstrations or tutorials that install their dependencies automatically.

Since its initial release, `davos` has found use in a variety of applications. In addition to managing computing environments for multiple ongoing research studies, `davos` is being used by both students and instructors in programming and methods courses such as Storytelling with Data [22] (an open course on data science, visualization, and communication) and Laboratory in Psychological Science [23] (an open course on experimental and statistical methods for psychology research) to simplify distributing lessons and submitting assignments, as well as in online demos such as `abstract2paper` [24]

336 (an example application of GPT-Neo [25, 26]) to share ready-to-run code
337 that installs dependencies automatically.

338 Our work also has several more subtle “advanced” use cases and potential
339 impacts. Whereas Python’s built-in `import` statement is agnostic to pack-
340 ages’ version information, `smuggle` statements (when combined with onion
341 comments) are version-sensitive. And because onion comments are parsed
342 at runtime, required packages and their specified versions are installed in a
343 just-in-time manner. Thus, it is possible in most cases to `smuggle` a specific
344 package version or revision even if a different version has already been loaded.
345 This enables more complex uses that take advantage of multiple versions of
346 a package within a single interpreter session. This could be useful in cases
347 where specific features are added or removed from a package across differ-
348 ent versions, or in comparing the performance or functionality of particular
349 features across different versions of the same package.

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

364 5. Conclusions

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

372 Author Contributions

373 **Paxton C. Fitzpatrick:** Conceptualization, Methodology, Software,
374 Validation, Writing - Original Draft, Visualization. **Jeremy R. Manning:**
375 Conceptualization, Resources, Validation, Writing - Review & Editing, Su-
376 pervision, Funding acquisition.

377 Funding

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

381 Declaration of Competing Interest

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

385 Acknowledgements

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

389 References

- 390 [1] G. van Rossum, Python reference manual, Department of Computer
391 Science [CS] (R 9525) (1995).
- 392 [2] Python Software Foundation, The Python Package Index (PyPI),
393 <https://pypi.org> (2003).
- 394 [3] conda-forge community, The conda-forge Project: Community-based
395 Software Distribution Built on the conda Package Format and Ecosys-
396 tem, <https://doi.org/10.5281/zenodo.4774217> (July 2015). doi:
397 [10.5281/zenodo.4774217](https://doi.org/10.5281/zenodo.4774217).
- 398 [4] N. Coghlan, D. Stufft, Version Identification and Dependency Specifica-
399 tion, PEP 440, Python Software Foundation (March 2013).
- 400 [5] B. Cannon, N. Smith, D. Stufft, Specifying minimum build system re-
401 quirements for python projects, PEP 518, Python Software Foundation
402 (May 2016).

- 403 [6] Anaconda, Inc., conda, <https://docs.conda.io> (2012).
- 404 [7] S. Eustace, Poetry: Python packaging and dependency management
405 made easy, <https://github.com/python-poetry/poetry> (December
406 2019).
- 407 [8] T. Kluyver, B. Ragan-Kelley, F. Pérez, B. Granger, M. Bussonnier,
408 J. Frederic, K. Kelley, J. Hamrick, J. Grout, S. Corlay, P. Ivanov,
409 D. Avila, S. Abdalla, C. Willing, Jupyter Notebooks – a publish-
410 ing format for reproducible computational workflows, in: F. Loizides,
411 B. Schmidt (Eds.), Positioning and Power in Academic Publishing: Play-
412 ers, Agents and Agendas, IOS Press, Netherlands, 2016, pp. 87–90.
413 [doi:10.3233/978-1-61499-649-1-87](https://doi.org/10.3233/978-1-61499-649-1-87).
- 414 [9] R. P. Goldberg, Survey of virtual machine research, Computer 7 (6)
415 (1974) 34–45.
- 416 [10] Y. Altintas, C. Brecher, M. Weck, S. Witt, Virtual Machine Tool,
417 CIRP Annals 54 (2) (2005) 115–138. [doi:https://doi.org/10.1016/](https://doi.org/10.1016/S0007-8506(07)60022-5)
418 [S0007-8506\(07\)60022-5](https://doi.org/10.1016/S0007-8506(07)60022-5).
- 419 [11] M. Rosenblum, VMware’s Virtual Platform: A virtual machine monitor
420 for commodity PCs, in: IEEE Hot Chips Symposium, IEEE, 1999, pp.
421 185–196.
- 422 [12] D. Merkel, Docker: lightweight linux containers for consistent develop-
423 ment and deployment, Linux Journal 239 (2) (2014) 2.
- 424 [13] G. M. Kurtzer, V. Sochat, M. W. Bauer, Singularity: Scientific contain-
425 ers for mobility of compute, PLoS One 12 (5) (2017) e0177459.
- 426 [14] F. Pérez, B. E. Granger, IPython: a system for interactive scientific
427 computing, Computing in science and engineering 9 (3) (2007) 21–29.
428 [doi:10.1109/MCSE.2007.53](https://doi.org/10.1109/MCSE.2007.53).
- 429 [15] G. van Rossum, J. Lehtosalo, L. Langa, Type Hints, PEP 484, Python
430 Software Foundation (September 2014).
- 431 [16] L. Torvalds, J. Hamano, Git: Fast version control system, [https://](https://git.kernel.org/pub/scm/git/git.git)
432 git.kernel.org/pub/scm/git/git.git (April 2005).
- 433 [17] K. Thompson, Programming Techniques: Regular expression search al-
434 gorithm, Communications of the ACM 11 (6) (1968) 419–422. [doi:](https://doi.org/10.1145/363347.363387)
435 [10.1145/363347.363387](https://doi.org/10.1145/363347.363387).

- [18] W. McKinney, Data Structures for Statistical Computing in Python, in: S. van der Walt, J. Millman (Eds.), Proceedings of the 9th Python in Science Conference, 2010, pp. 56–61. doi:10.25080/Majora-92bf1922-00a.
- [19] 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.
- [20] G. Varoquaux, Joblib: Computing with Python functions, <https://github.com/joblib/joblib> (July 2010).
- [21] 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.
- [22] J. R. Manning, Storytelling with Data, <https://github.com/ContextLab/storytelling-with-data> (June 2021). doi:10.5281/zenodo.5182775.
- [23] J. Manning, ContextLab/experimental-psychology: v1.0 (Spring, 2022), <https://github.com/ContextLab/experimental-psychology/tree/v1.0> (May 2022). doi:10.5281/zenodo.6596762.
- [24] J. R. Manning, abstract2paper, <https://github.com/ContextLab/abstract2paper> (June 2021). doi:10.5281/zenodo.7261831.
- [25] L. Gao, S. Biderman, S. Black, L. Golding, T. Hoppe, C. Foster, J. Phang, H. He, A. Thite, N. Nabeshima, S. Presser, C. Leahy, The Pile: An 800GB Dataset of Diverse Text for Language Modeling, arXiv preprint arXiv:2101.00027 (2020).
- [26] S. Black, L. Gao, P. Wang, C. Leahy, S. Biderman, GPT-Neo: Large Scale Autoregressive Language Modeling with Mesh-Tensorflow, <http://github.com/eleutherai/gpt-neo> (2021).