`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 $\geq$ 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	`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.

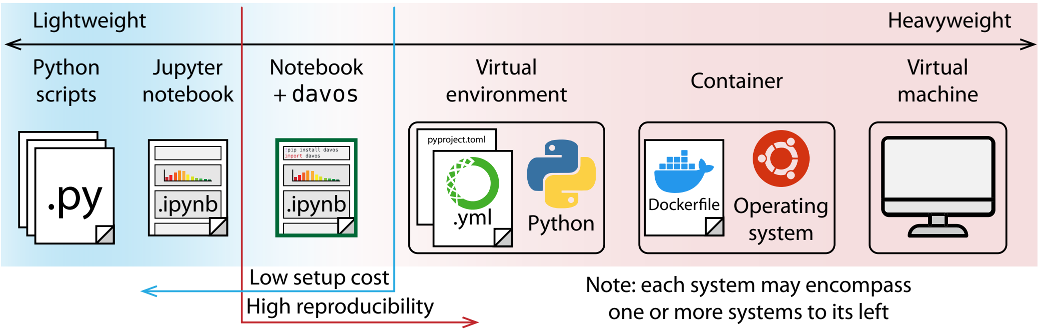

Figure 1: **Systems for sharing code within the Python ecosystem.** From left to right: plain-text **Python scripts** (`.py` files) provide the most basic "system" for sharing raw code. Scripts may reference external libraries, but those libraries must be manually installed on other users' systems. Further,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

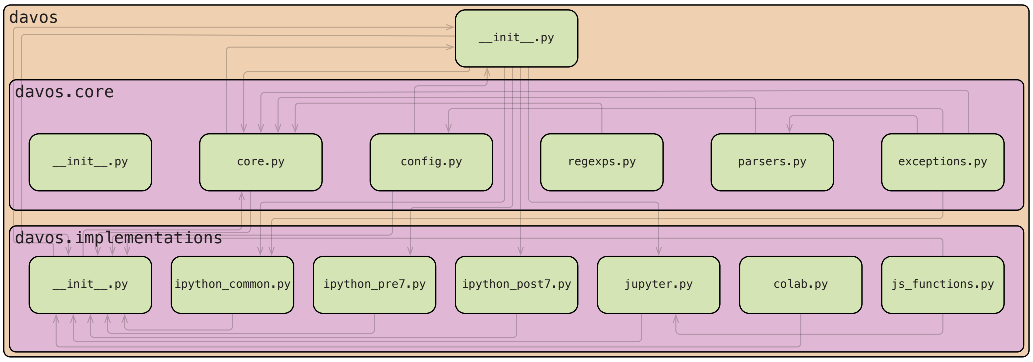

Figure 2: **Package structure.**

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 proper implementations for the current notebook environment in a single, consistent location. An additional benefit of this design pattern is that it allows both maintainers and users to easily extend `davos` to support new, updated, or custom notebook variants by creating a new `davos.implementations` module that defines its own version of each helper function, modified as much or little as necessary.

2.2. Software functionalities

2.2.1. The `smuggle` statement

Functionally, importing `davos` in an IPython notebook enables 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`" 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:

```python
# 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 that `davos` installs the proper distribution if the smuggled package can't be found locally:

```python
# 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:

```python
# 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.):

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

davos processes onion comments internally before forwarding arguments to the installer program. In addition to preventing onion comments from being used as a vehicle for shell injection attacks, this enables davos to take certain logical actions when particular arguments are passed. For example, each of the -I/--ignore-installed, -U/--upgrade, and --force-reinstall flags will cause davos to skip searching for a smuggled package locally before installing 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
```

Similarly, passing --no-input will temporarily enable davos's non-interactive mode (see Section 2.2.3), and installing a smuggled package into <dir> with --target <dir> will cause davos to prepend dir to the module search path (i.e., sys.path), if necessary, so the package can be imported.

2.2.3. The davos config

The davos config provides a simple, high-level interface for controlling various aspects of davos's behavior. After importing davos, the davos.config object (a singleton) exposes a number of configurable options as attributes that can be assigned different values, checked at runtime, and displayed in the notebook cell output (see Fig. 4 for example usage). These include:

- .active: This option allows users to disable davos functionality (i.e., support for smuggle statements and onion comments) for subsequent notebook cells by setting its value to False. davos can be re-enabled at any time by setting this option to True (the default when davos is first imported). See Section 2.3 for additional info.

- .auto_rerun: This option controls how davos behaves when attempting to smuggle a new version of a package that was previously imported and cannot be reloaded. This can happen if the package includes extension modules that dynamically link C or C++ objects to the Python interpreter itself, and the code that generates those objects was changed between the old and new versions. If this option is set to True, davos will automatically restart the notebook kernel and rerun all code up to (and including) the current smuggle statement. If False (the deafult), davos will instead issue a warning, pause execution, and prompt the user with buttons to either restart and rerun the notebook, or continue running with the previously imported package version until the next time the kernel is restarted manually. (Note: not configurable in Google Colaboratory).

- `.confirm_install`: If `True` (default: `False`), `davos` will require user confirmation (`[y]es/[n]o` input) before installing a smuggled package.

- `.noninteractive`: Setting this option to `True` (default: `False`) enables non-interactive mode, in which all user input and confirmation is disabled. Note that in non-interactive mode, the `confirm_install` option is set to `False`, and if `auto_rerun` is `False`, `davos` will throw an error if a smuggled package cannot be reloaded, rather than prompting the user.

- `.pip_executable`: This option's value specifies the path to the `pip` executable used to install smuggled packages. The default is programmatically determined from the Python environment and falls back to `sys.executable -m pip` if no executable can be found.

- `.suppress_stdout`: If `True` (default: `False`), suppress all unnecessary output issued by both `davos` and the installer program. This can be useful when smuggling packages that need to install many dependencies and/or generate extensive output. If the installer program throws an error, both the stdout and stderr streams will be displayed with the traceback.

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

2.3. Implementation details

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

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

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

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

```
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})
```

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

3. Illustrative Examples

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

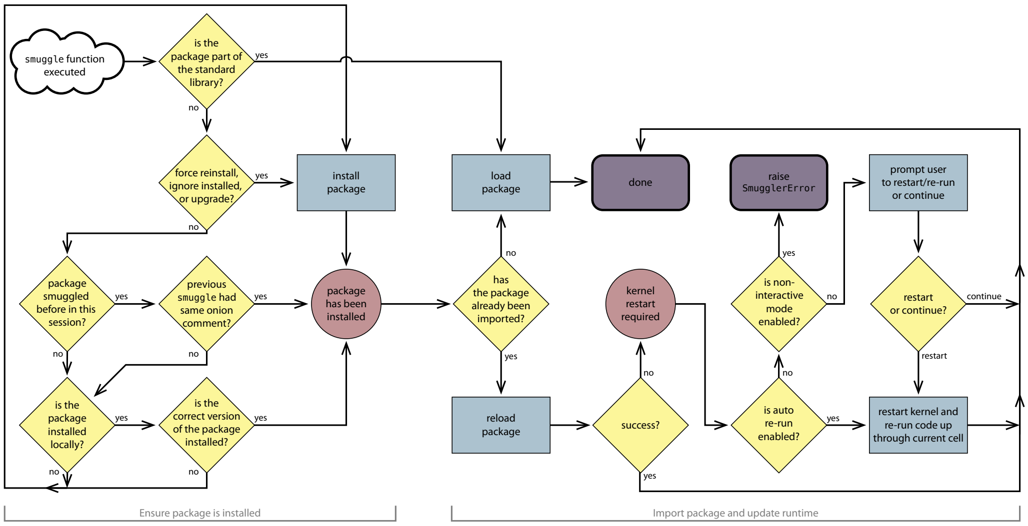

Figure 3: `smuggle()` **function logic.**

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 does not need or benefit from an onion comment. 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-`

```
import davos

from os.path smuggle is_file
smuggle joblib                          # pip: joblib<=1.2.0

davos.config.auto_rerun = True
smuggle numpy as np                     # pip: numpy==1.21.6

if not is_file("~/datasets/data-new.csv"):
    smuggle pandas as pd                # pip: pandas<0.25.0
    tmp_data = pd.read_pickle("~/datasets/data-old.pkl")
    tmp_data.to_frame().to_csv("~/datasets/data-new.csv")

smuggle pandas as pd                    # pip: pandas==1.3.5

davos.configure(auto_rerun=False, suppress_stdout=True, noninteractive=True)
smuggle tensorflow as tf                # pip: tensorflow==2.9.2
from umap smuggle UMAP                  # pip: umap-learn[plot,parametric_umap]==0.5.3
davos.configure(suppress_stdout=False, noninteractive=False)

smuggle matplotlib.pyplot as plt        # pip: matplotlib==3.5.3
smuggle seaborn as sns                  # pip: seaborn==0.12.1
smuggle quail                           # pip: git+https://github.com/myfork/quail@6c847a4

davos.config.pip_executable = "~/envs/nb-server/bin/pip"
smuggle widgetsnbextension as _     # pip: widgetsnbextension==3.5.2
davos.config.pip_executable = "~/envs/nb-kernel/bin/pip"
smuggle ipywidgets                  # pip: ipywidgets==7.6.5

from tqdm.notebook smuggle tqdm         # pip: tqdm==4.62.3

data = pd.read_csv("~/datasets/data-new.csv", index_col=[0, 1])
smuggle sklearn                         # pip: scikit-learn<0.22.0
transformer = joblib.load("~/models/text-transformer.joblib")
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 libraries rely heavily on custom C data types, loading the particular versions from the onion comments 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 imported and cached 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_rerun option before smuggling NumPy will help 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 persisting data in Python, allowing users to save, share, and load arbitrary objects, though with an important caveat: In order to successfully "unpickle" (i.e., load and restore) a "pickled" object, its class must be defined in and importable from the same module as when it was saved. The Panel class was removed from pandas in July, 2019 (v0.25.0), meaning that the dataset can be loaded only if the user's notebook uses a pandas version released prior to that change. However, more recent 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 for data structures (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 version of pandas 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 will then install a newer pandas version (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 will require restarting the notebook kernel. When davos determines this, it will automatically restart the kernel, re-run the notebook up through the current cell, and also re-queue any subsequent cells that were queued before the restart was triggered. This second time the code is run (as well as in any future runs), lines 10–12 will be skipped since the CSV-formatted dataset has already been created, and the `smuggle` statements up through line 14 will execute (virtually) as fast as regular `import` statements since the required package versions have already been installed.

Line 16 then uses the `davos.configure()` function to disable the `auto_rerun` option and simultaneously enable two other options: `suppress_stdout` and `noninteractive`. With these options enabled, lines 17–18 `smuggle` `TensorFlow` [22], a powerful end-to-end platform for building and working with machine learning models, and `UMAP` [23], a library that implements a family of related manifold learning techniques. The onion comment in line 18 also specifies that `UMAP` should be installed with the optional requirements needed for its "plot" and "parametric_umap" features. Together, these two packages depend on 36 other unique packages, most of which have dependencies of their own. And if many of these are not already installed in the user's environment, lines 17–18 could take multiple minutes to run. Enabling the `noninteractive` option ensures the installation process won't require any input from the user to finish successfully, allowing them to simply ignore the notebook during this time and either work in another browser tab or application, or step away from their computer entirely. Additionally, installing more than three dozen packages could result in the installer program writing an extremely large amount of standard output text to the notebook cell's output area. Enabling `suppress_stdout` hides the output from these two `smuggle` statements so that other potentially important output (e.g., from previous `smuggle` statements) isn't buried and lost. Importantly, if either line fails for any reason, both the stdout and stderr streams will be shown alongside the regular Python traceback.

After re-enabling these two options (line 19), the user next smuggles specific versions of three plotting libraries: `Matplotlib` [24], `seaborn` [25], and `Quail` [26] (lines 21–23). Because the first two are requirements of `UMAP`'s optional "plot" feature, they will have already been installed by line 18, though possibly as different versions than those specified in the onion comments on lines 21 and 22. If the installed and specified versions are the same, these `smuggle` statements will function like regular `import` statements to load the libraries into the notebook namespace. If they differ, `davos` will download the requested versions in place of the installed versions before doing so. It's also worth noting that while loading `UMAP` in line 18 will not result in either `Matplotlib` or `seaborn` being imported internally due to the specific layout of the `UMAP` package, if it had, `davos` would also gracefully replace the cached versions of the two packages so that the smuggled versions are loaded properly. And (as with all other `smuggle` statements in Figure 4), if for any reason the specified package versions are incompatible with any other packages (e.g., `UMAP`), the installer program would display a warning to this effect in the notebook.

The third plotting library smuggled, `Quail`, is also pinned to a precise version with an onion comment, though in a slightly different manner than other packages in Figure 4. Line 23 installs and loads `Quail` from the user's fork ("myfork") of the package's GitHub repository, at a specific point in its revision history (i.e., the commit whose short hash is "6c847a4"). This can serve as a useful way to access a version of a package the user has customized, patched, or somehow modified for this specific use case, while still allowing them to further modify their fork of the package in the future without affecting the version used in these analyses.

Next, the user smuggles a pair of packages that extend the Jupyter notebook interface with various interactive JavaScript widgets. The `ipywidgets` [27] package provides an API for creating these widgets with Python code, and the `widgetsnbextension` package provides the machinery for the notebook frontend to display them. Thus, `ipywidgets` must be installed in the same environment as the IPython kernel, and `widgetsnbextension` must be installed in the environment that houses the Jupyter notebook server. In the simplest case, these two environments are the same; however, a common approach to using Jupyter notebooks with multiple different Python environments (e.g., for different tasks or projects) is to run the notebook server from a "base environment," with additional environments each providing their own separate, interchangeable IPython kernels. (While possible to detect programmatically, Figure 4 assumes this setup to simplify the example code.) To handle this multi-environment installation, line 25 temporarily sets the `pip` executable used to install smuggled packages to that of the notebook server's environment before smuggling `widgetsnbextension` (line 26). Since there is no need to actually load this package for later use in the analysis code, it is aliased as a single underscore—by convention, a "throwaway" variable in Python. In line 27, the user changes the `pip` executable back to that of the notebook kernel environment (its original value) before installing `ipywidgets` and subsequent packages. With these two packages installed, line 30 smuggles `tqdm` [28], which provides convenient progress bars for long-running code. In Jupyter notebooks, the `tqdm.notebook` module can be imported to enable prettier progress bars displayed via `ipywidgets`, if that package is installed and importable. Therefore, to take advantage of this feature, it was important to `smuggle tqdm` after ensuring the `ipywidgets` package was available.

Finally, the user loads in the re-formatted dataset (line 32) and pre-trained model (line 34) they'll use in their analyses. But in accessing the latter, the user encounters an issue similar to the one they initially faced with the dataset. This hypothetical semantic model was provided as a `scikit-learn` "`Pipeline`" object that allows the user to pass text data through two pre-trained transformers, in series: a `CountVectorizer` instance and a `LatentDirichletAllocation` instance (i.e., a topic model [29]). The model was saved by its original creator using the `joblib` library, as `scikit-learn`'s documentation recommends, with the caveat that because `joblib` uses the `pickle` protocol internally, the ability to save and load pre-trained models is not guaranteed across versions due to `pickle`'s requirement that referenced classes be importable from the same location when saved and loaded. This issue affects the user's model by way of the `LatentDirichletAllocation` class—when the `Pipeline` object was created (again, in June, 2019, using `scikit-learn` v0.21.3), the class's definition lived in the `sklearn.decomposition.online_lda` module. However, in version 0.22.0 (released November, 2019), that module was renamed to "`_online_lda`," and in v0.22.1 (January, 2020) it was again renamed to "`_lda`" (which has remained its name since then). Thus, in order to successfully load the model that includes the pre-trained `LatentDirichletAllocation` instance, in line 33, the user first smuggles a version of `scikit-learn` prior to v0.22.0 (i.e., before the first time the module's name was changed). Once the model is loaded and reconstructed in memory from a compatible package version (line 34), the user then upgrades to a newer version of `scikit-learn` for use elsewhere in their analyses. When line 35 is run, `davos` will replace the existing `scikit-learn` version with v1.1.3 in both the user's local package environment and in memory without interruption, such that "`sklearn`" will reference the new version in any code below. However, this will not affect the already compiled code object stored in the "`transformer`" variable, meaning that the pre-trained model will be usable simultaneously.

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 [30] (an open course on data science, visualization, and communication) and Laboratory in Psychological Science [31] (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` [32] (an example application of GPT-Neo [33, 34]) to share ready-to-run code that installs dependencies automatically.

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

A second advanced use case is in providing a proof-of-concept of how one can add new "keyword-like" operators to the Python language by leveraging notebooks' error-handling mechanisms. This could lead to exciting new tools that, like `davos`, extend the Python language in useful ways within notebook-based environments. We note that our approach to adding the `smuggle` keyword to Python when `davos` is imported into a notebook-based environment also has the potential to be exploited for more nefarious purposes. For example, a malicious user could use a similar approach (e.g., in a different library) to substantially change a notebook's functionality by adding new *unexpected* keyword-like objects (e.g., based around common typos). This could lead to difficult-to-predict changes in a notebook's behavior once the malicious library was imported. This highlights an important reason why security-conscious users would be well-served to only make use of libraries from trusted sources, or whose code is publicly available for review.

5. Conclusions

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

Author Contributions

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

Funding

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

Declaration of Competing Interest

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

Acknowledgements

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

References

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

[2] Python Software Foundation, The Python Package Index (PyPI), https://pypi.org (2003).

[3] conda-forge community, The conda-forge Project: Community-based Software Distribution Built on the conda Package Format and Ecosystem, https://doi.org/10.5281/zenodo.4774217 (July 2015). doi:10.5281/zenodo.4774217.

[4] N. Coghlan, D. Stufft, Version Identification and Dependency Specification, PEP 440, Python Software Foundation (March 2013).

[5] B. Cannon, N. Smith, D. Stufft, Specifying minimum build system requirements for python projects, PEP 518, Python Software Foundation (May 2016).

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

[7] S. Eustace, Poetry: Python packaging and dependency management made easy, https://github.com/python-poetry/poetry (December 2019).

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

[9] R. P. Goldberg, Survey of virtual machine research, Computer 7 (6) (1974) 34–45.

[10] 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.

[11] M. Rosenblum, VMware's Virtual Platform: A virtual machine monitor for commodity PCs, in: IEEE Hot Chips Symposium, IEEE, 1999, pp. 185–196.

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

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

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

[15] G. van Rossum, J. Lehtosalo, Ł. Langa, Type Hints, PEP 484, Python Software Foundation (September 2014).

[16] L. Torvalds, J. Hamano, Git: Fast version control system, https://git.kernel.org/pub/scm/git/git.git (April 2005).

[17] K. Thompson, Programming Techniques: Regular expression search algorithm, Communications of the ACM 11 (6) (1968) 419–422. doi: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] M. Abadi, A. Agarwal, P. Barham, E. Brevdo, Z. Chen, C. Citro, G. S. Corrado, A. Davis, J. Dean, M. Devin, S. Ghemawat, I. Goodfellow, A. Harp, G. Irving, M. Isard, Y. Jia, R. Jozefowicz, L. Kaiser, M. Kudlur, J. Levenberg, D. Mané, R. Monga, S. Moore, D. Murray, C. Olah, M. Schuster, J. Shlens, B. Steiner, I. Sutskever, K. Talwar, P. Tucker, V. Vanhoucke, V. Vasudevan, F. Viégas, O. Vinyals, P. Warden, M. Wattenberg, M. Wicke, Y. Yu, X. Zheng, TensorFlow: Large-Scale Machine Learning on Heterogeneous Systems, software available from tensorflow.org (2015).
URL https://www.tensorflow.org/

[23] L. McInnes, J. Healy, N. Saul, L. Großberger, UMAP: Uniform Manifold Approximation and Projection, Journal of Open Source Software 3 (29) (2018) 861. doi:https://doi.org/10.21105/joss.00861.

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

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

[26] A. C. Heusser, P. C. Fitzpatrick, C. E. Field, K. Ziman, J. R. Manning, Quail: a Python toolbox for analyzing and plotting free recall data, Journal of Open Source Software 10.21105/joss.00424 (2017).

[27] J. Frederic, J. Grout, Jupyter Widgets Contributors, ipywidgets: Interactive Widgets for the Jupyter Notebook, https://github.com/jupyter-widgets/ipywidgets (August 2015).

[28] C. da Costa-Luis, S. K. Larroque, K. Altendorf, H. Mary, richardsheridan, M. Korobov, N. Raphael, I. Ivanov, M. Bargull, N. Rodrigues, G. Chen, A. Lee, C. Newey, CrazyPython, JC, M. Zugnoni, M. D. Pagel, mjstevens777, M. Dektyarev, A. Rothberg, A. Plavin, D. Panteleit, F. Dill, FichteFoll, G. Sturm, HeoHeo, H. van Kemenade, J. McCracken, MapleCCC, M. Nordlund, tqdm: A Fast, Extensible Progress

Bar for Python and CLI, https://github.com/tqdm/tqdm (September 2022). doi:10.5281/zenodo.595120.

[29] D. M. Blei, A. Y. Ng, M. I. Jordan, Latent dirichlet allocation, Journal of Machine Learning Research 3 (2003) 993–1022.

[30] J. R. Manning, Storytelling with Data, https://github.com/ContextLab/storytelling-with-data (June 2021). doi:10.5281/zenodo.5182775.

[31] 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.

[32] J. R. Manning, abstract2paper, https://github.com/ContextLab/abstract2paper (June 2021). doi:10.5281/zenodo.7261831.

[33] 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).

[34] 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).