`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)

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—or 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.

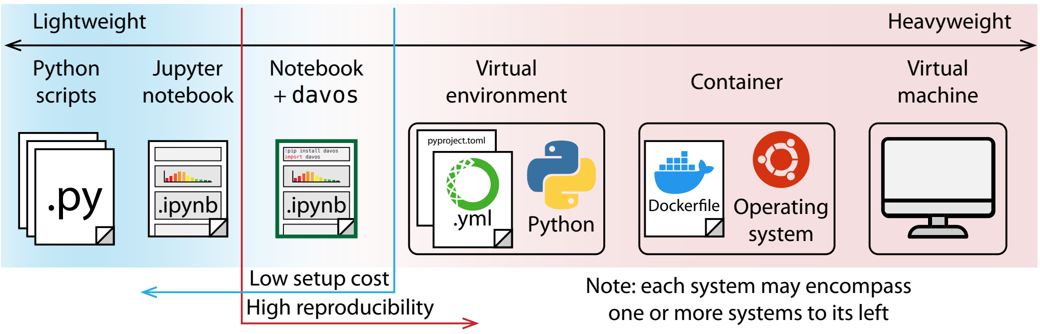

Figure 1: **Systems for sharing code within the Python ecosystem.** From left to right: plain-text **Python scripts** (`.py` files) provide the most basic "system" for sharing raw code. Scripts may reference external libraries, but those libraries must be manually installed on other users' systems. Further,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

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 referred to as "the Onion Knight" from the series *A Song of Ice and Fire* by George R. R. Martin [14]. The `smuggle` keyword-like operator implemented in `davos` is a play on Python's `import` keyword: whereas importing can bring a package into the Python workspace within the existing rules and frameworks provided by the Python language, "smuggling" provides an alternative that expands the scope and reach of "importing." Like the character Davos Seaworth (who became famous for smuggling onions through a blockade on his homeland), we use "onion" comments to precisely control how packages are smuggled into the Python workspace.

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 [15] 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

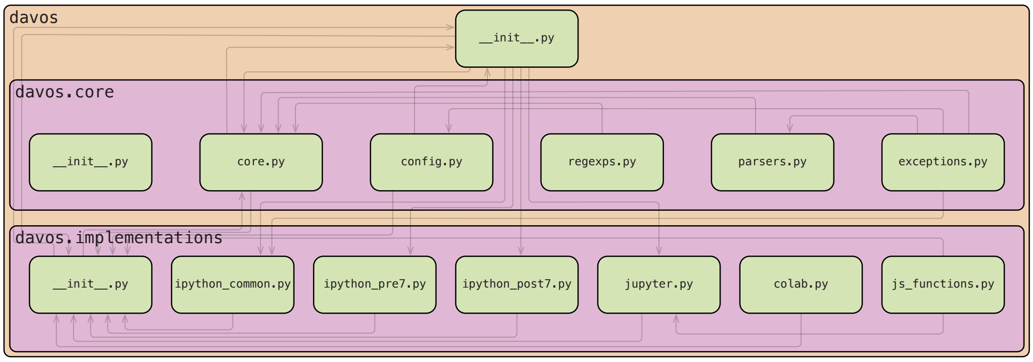

Figure 2: **Package structure.** The `davos` library comprises two interdependent subpackages. The `davos.core` subpackage includes modules for parsing `smuggle` statements and onion comments, installing and validating packages, and configuring `davos`'s behavior. The `davos.implementations` subpackage includes environment-specific modifications and features that are needed to support the core functionality across different notebook-based environments. Individual `.py` files are represented by lime rounded rectangles, and arrows denote dependencies (each arrow points to a file that import objects defined or specified at the arrow's source).

notebook environment in a single, consistent location. An additional benefit of this design is that it allows maintainers, developers, and users to extend `davos` to support new, updated, or custom notebook variants by creating new `davos.implementations` modules that define their own versions of each helper function, modified from existing implementations as needed.

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` keyword-like object can be used as a drop-in replacement for Python's built-in `import` keyword 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 [16], and are designed to make managing packages with `davos` intuitive and familiar. To construct an onion comment, users provide the name of the installer program (e.g., `pip`) and the same arguments one would use to manually install the package as desired via the command line:

```
# enable smuggle statements
import davos

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

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

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

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

Because onion comments may be constructed to specify any aspect of the installer's behavior, they provide a mechanism for precisely controlling how, where, and when smuggled packages are installed. Critically, 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 information 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 [17] 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
```

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 design provides the user with additional control over which packages are imported. 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 a custom directory (<dir>) using the --target <dir> flag 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 object*

The davos config object provides a high-level interface for controlling various aspects of davos's behavior. After importing davos, the davos.config object (a singleton) exposes configurable options as attributes that can be modified, checked at runtime, or displayed (see Sec. 3 for an illustrative example or Sec. 2.3 for implementation details and additional information). These include:

- .active: This attribute controls whether support for smuggle statements and onion comments) is enabled (True) or disabled (False). When davos is first imported), the .active attribute is set to True.

- `.auto_rerun`: This attribute 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 if the code that generates those objects was changed between the previously imported and to-be-smuggled versions. If this attribute 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 default), `davos` will instead issue a warning, pause execution, and prompt the user 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 that, as of this writing, the `.auto_rerun` attribute is not supported in Google Colaboratory notebooks.

- `.confirm_install`: If `True` (default: `False`), `davos` will require user confirmation before installing a smuggled package that do not yet exist in the user's environment.

- `.noninteractive`: Setting this attribute to `True` (default: `False`) enables non-interactive mode, in which all user interactions (prompts and dialogues) are disabled. Note that in non-interactive mode, the `confirm_install` option is set to `False`. If `auto_rerun` is `False` while in non-interactive mode, `davos` will raise an exception if a smuggled package cannot be reloaded, rather than prompting the user.

- `.pip_executable`: This attribute'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 this attribute is set to `True` (default: `False`), `davos` suppresses printed (console) outputs from itself and the installer program. This can be useful when smuggling packages that need to install many dependencies and/or generate extensive output. However, if the installer program throws an error, both the stdout and stderr streams will still be displayed along with a stack trace.

The top-level `davos` namespace also defines convenience functions for setting and checking whether `davos` is active (`davos.activate()`; `davos.deactivate()`; `davos.is_active()`) as well as the `davos.configure()` function, which allows setting multiple configuration options simultaneously.

2.3. Implementation details

Although `davos` is designed to `appear` to add a new Python keyword to a notebook's vocabulary, this illusion is actually a consequence of several "hacks" that make use of the IPython backend for processing and executing notebook code. Specifically, when `davos` is first imported, or when it activated after being set to an inactive state, two actions are triggered. First, the `smuggle()` function is injected into the IPython user namespace. Second, 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` uses this process to transform `smuggle` statements into syntactically valid Python code. The `davos` parser uses a regular expression 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 carries out `davos`'s central logic by determining whether the smuggled package must be installed, carrying out the installation 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 Example

Across different versions of a given package, functions may be changed, depreciated, added, or renamed. In addition to changing the behaviors of

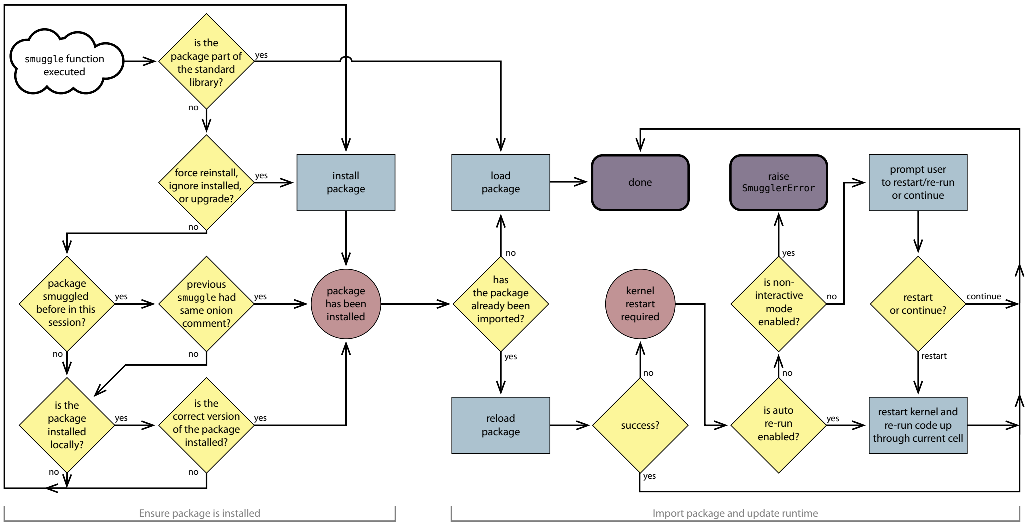

Figure 3: `smuggle()` **function algorithm.** At a high level, the `smuggle()` function may be conceptualized as following two basic steps. First (left), `davos` ensures that the correct version of the desired package has been installed, carrying out the installation automatically if needed. Second (right), `davos` imports the package and updates the current runtime environment.

active computations, saved objects created using one version of a package may be incompatible with another version of the same package. For example, the popular `pandas` [18] library prior to version 0.25.0 included the `Panel` datatype for storing 3-dimensional arrays. Since version 0.20.0, however, the `Panel` datatype has been depreciated, and the datatype was removed entirely in version 0.25.0. If one had saved a dataset in a `Panel` object (created using an older version of `pandas`), e.g., by serializing the `Panel` and saving it to disk using, `pickle`, that dataset could not be read by any version of `pandas` from 0.25.0 or beyond. These incompatibilities are not limited solely to traditional forms of data. For example, saved model states and other objects may reference functions, attributes, classes, and other objects that are not present across all versions of their associated library.

The example provided in Figure 4 demonstrates how the `davos` library can be used to circumvent these incompatibilities by carefully controlling which versions of each package are used in different parts of the notebook. The example shows how a data and model saved using since-depreciated functionality in the `pandas` and `scikit-learn` [19] packages may be loaded in (using older versions of each package) and manipulated or analyzed (using more recent versions of each package).

After importing `davos` (line 1), the user first smuggles two utilities for

```python
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.** Snippets from the example are also excerpted in the main text of Section 3.

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, a *future* release might introduce some breaking change. The onion comment in line 4 helps ensure the analysis notebook continues to run properly in the future, by limiting allowable versions to those already released when the code was written:

```
1  import davos
2
3  from os.path smuggle is_file
4  smuggle joblib                      # pip: joblib<=1.2.0
```

Line 6 then uses the `davos.config` object to enable `davos`'s `auto_rerun` 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 had been previously imported during the same interpreter session (see Section 2.2.3).

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

Setting the `auto_rerun` attribute to `True` also benefits the (potential) installation of `pandas` on the next lines:

```
 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.pkl")
12      tmp_data.to_frame().to_csv("~/datasets/data-new.csv")
13
14  smuggle pandas as pd                # pip: pandas==1.3.5
```

If we suppose that the data contained in `data-old.pkl` is stored in a `Panel` object (i.e., created in a version of `pandas` prior to 0.25.0), then we would not be able to load in that object with newer versions of `pandas`. Line 10 ensures that an older version of `pandas` will be imported, enabling the data to be read in (and saved out to a `.csv` file, which is compatible with newer versions of `pandas`).

Newer versions of `pandas` have brought substantial performance improvements, added new functionality, fixed bugs, etc. Although the original dataset had to be read in using an older version of the package, we can take advantage of those newer changes using a new `smuggle` statement and onion comment on line 14 (which specifies that version 1.3.5 should be imported). Since `pandas` had already been imported into the current workspace (on line 10), the runtime must be restarted in order to gain access to the newer version of `pandas`. The `.auto_rerun` flag set on line 6 enables this process to happen automatically, and saving out the dataset to a `.csv` file in lines 9–12 ensures that the older version of `pandas` does not need to be reinstalled.

Next, line 16 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 several minutes to run. Enabling the `noninteractive` option ensures that the installation will continue automatically without user input during that time. Enabling `suppress_stdout` also suppresses console outputs from the installer that might otherwise distract from other more important outputs.

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

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 standard `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.

Line 23 uses an onion comment to specify that `Quail` should be installed directly from a specific GitHub commit (`6c847a4`). This ability to install packages directly from GitHub repositories can enable developers to use forked or modified versions of other libraries in their notebooks, even if those versions have not been officially released.

```
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
```

In lines 25–28, we demonstrate another aspect of `davos`'s functionality that supports more advanced installation scenarios. 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.

```
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
```

A complication is that `ipywidgets` must be installed in the same environment as the IPython kernel, whereas `widgetsnbextension` must be installed in the environment that houses the Jupyter notebook server. In standard setups, these two environments are the same. However, a common "advanced" approach is to run the notebook server from a base environment, with additional environments each providing their own separate, interchangeable IPython kernels. To accomodate this multi-environment scenario, on lines 25 and 27 we control which environments each package should be installed to. Once these two packages are installed and imported, line 30 smuggles `tqdm` [28], which display progress bars to provide status updates for running code. In Jupyter notebooks, the `tqdm.notebook` module can be imported to enable aesthetically pleasing progress bars that are 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.

Next, we load in the reformatted dataset (line 32) and pre-trained model (line 34) that we wish to use in our analysis. In our hypothetical example, we can suppose that the model was provided as a `scikit-learn` `Pipeline` object that passes data through two pretrained models in succession. First, a trained `CountVectorizer` instance converts text data to an array of word counts. Second, the word counts are passed to a topic model [29] using a pretrained `LatentDirichletAllocation` instance.

Let us suppose that the `Pipeline` object had been saved by its original creator using the `joblib` library, as `scikit-learn`'s documentation recommends. Because `joblib` uses the `pickle` protocol internally, the ability to save and load pre-trained models is not guaranteed across different scikit-learn versions. For example, suppose that the Pipeline object was created using scikit-learn v0.21.3). In that version of scikit-learn, the LatentDirichletAllocation class is defined in sklearn.decomposition.online_lda. However, in version 0.22.0, that module was renamed to _online_lda, and in versions 0.22.1 and higher, the name has been reverted back to _lda.

In order to correctly load the model that includes the pretrained 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 relevant module's name was changed). Once the model is loaded and reconstructed in memory from a compatible package version (line 34), we upgrade to a newer version of scikit-learn in line 35. Taken together, the code in Figure 4 shows how davos can enable users to load in data and models that are incompatible with newer versions of pandas and scikit-learn, but still *analyze* and the data and model output using the latest approaches and implementations.

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 use the davos library to ensure their notebooks will run correctly 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 (e.g., see Section 3). 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 more subtle impact of our work is in providing a proof-of-concept of how the ability to add new "keyword-like" operators to the Python language could be specifically useful to researchers. With `davos`, we accomplish this by leveraging IPython notebooks' internal code parsing and execution machinery. We note that, while other popular packages similarly use these mechanisms to providing notebook-specific functionality (e.g., [24, 35]), this approach also has the potential to be exploited for more nefarious purposes. For example, a malicious user could design a Python library that, when imported, substantially changes the notebook's functionality by adding new *unexpected* keyword-like objects (e.g., based around common typos). We also note that this implementation approach means `davos`'s functionality is currently restricted to IPython notebook environments, as would be that of any potential similar tools that enable user-defined keywords. However, there has been early-stage discussion of providing this sort of syntactic customizability as a core feature of the Python language, including a draft proposal [36]. In addition to enabling `davos` to be extended for use outside of notebooks, this could lead to exciting new tools that, like `davos`, extend the Python language in useful and more secure ways.

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] G. R. R. Martin, A song of fire and ice, HarperVoyager, London, England, 2012.

[15] 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.

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

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

[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).

[35] A. C. Heusser, K. Ziman, L. L. W. Owen, J. R. Manning, HyperTools: a Python toolbox for gaining geometric insights into high-dimensional data, Journal of Machine Learning Research 18 (152) (2018) 1–6.

[36] M. Shannon, Syntactic Macros, Draft PEP 638, Python Software Foundation (September 2020).