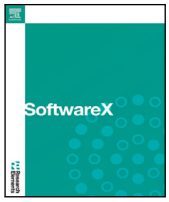
SoftwareX 25 (2024) 101614

Contents lists available at ScienceDirect

SoftwareX

journal homepage: www.elsevier.com/locate/softx



Original software publication

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, United States of America

ARTICLE INFO

Keywords:
Reproducibility
Open science
Python
Jupyter notebook
Google colaboratory
Package management

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 both sharing and engaging with reproducible science. Here, we propose a lighter-weight solution: the `Davos` package. When used in combination with a notebook-based Python project, `Davos` provides a mechanism for specifying the correct versions of the project's dependencies directly within the code that requires them, and automatically installing them in an isolated environment when the code is run. The `Davos` package further ensures that these 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.

Code metadata

Current code version	v0.2.3
Permanent link to code/repository used for this code version	https://github.com/ElsevierSoftwareX/SOFTX-D-22-00400
Code Ocean compute capsule	N/A
Legal Code License	MIT
Code versioning system used	Git
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
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, Kaggle, IDE-based notebook editors, IPython shell.
Link to developer documentation/manual	https://github.com/ContextLab/davos#readme
Support email for questions	contextualdynamics@gmail.com

1. Motivation and significance

The same computer code may not behave identically under different circumstances. For example, when code depends on external packages, different versions of those packages 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].

* Corresponding author.
E-mail address: Jeremy.R.Manning@Dartmouth.edu (Jeremy R. Manning).

https://doi.org/10.1016/j.softx.2023.101614
Received 30 November 2022; Received in revised form 27 October 2023; Accepted 7 December 2023
Available online 19 December 2023

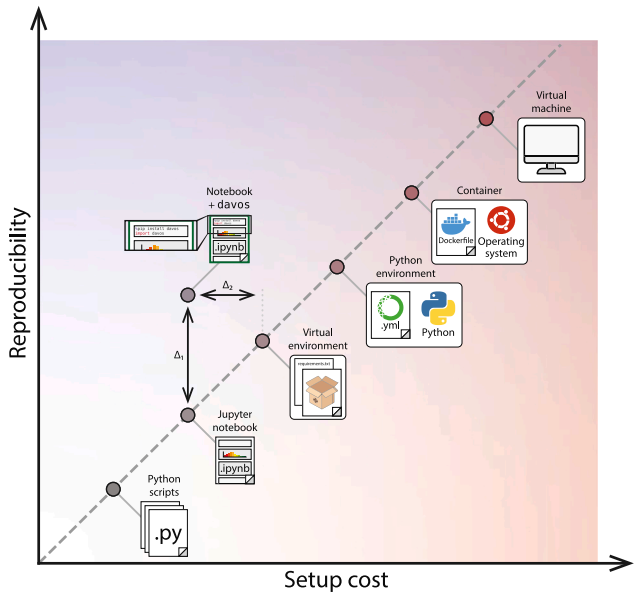


Fig. 1. Systems for ensuring code reproducibility within the Python ecosystem. The *x*-axis denotes the "burden" placed on users to install and configure the given system (systems placed further to the right, that fall within the redder shading, impose a higher setup cost on the user). The *y*-axis denotes the degree to which the system guarantees that the code will run similarly for different users (systems placed higher up, that fall within the bluer shading, offer stronger guarantees). From left to right, bottom to top: plain-text **Python scripts** (`.py` files) provide the most basic "system" for sharing raw code. Scripts may reference external packages, but those packages must be manually installed on other users' systems. Further, any checking needed to verify that the correct versions of those packages 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 package** is imported into a Jupyter notebook, the notebook's functionality is extended to automatically install any required external packages (at their correct versions, when specified) into an isolated directory kept separate from the user's existing set of packages. **Virtual environments** similarly allow users to isolate installed packages in a specific directory, and can be populated with packages from a shared dependency specification file (e.g., a `requirements.txt` or `pyproject.toml` [6]). **Python environments** (again, shared in the form of a dependency specification file; e.g., an `environment.yml`) extend this idea to allow installing a specific version of Python itself. However, both virtual environments and Python environments require users to create, populate, and manage environments manually. This typically entails distributing a configuration file (e.g., a `requirements.txt`, `pyproject.toml` [6], or `environment.yml` file) that specifies all project dependencies (including version numbers) alongside the primary code base. Users can then install a third-party tool [e.g., 7–9] 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` [10]) 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. Δ_1 represents the increased reproducibility `Davos` provides over standard Jupyter notebooks for no greater setup cost. Δ_2 represents `Davos`'s lower setup cost compared to standard virtual environments despite more stringently ensuring reproducibility.

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 [5].

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 [11] that comprise a mix of text, executable code, and embedded media. Notebooks may call or import external scripts or packages—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 [12–14] 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., 10,15] 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. Shareable Python environments [e.g., 9] also provide a computing environment that is largely separated from the user's main environment. They incorporate (i.e., automatically install) a copy of Python and the target software's dependencies, but do not specify or reproduce an operating system for the runtime environment. Virtual environments [e.g., 7,8] work similarly, but reuse an existing copy of Python rather than bundling their own. Each of these systems (virtual machines, containers, Python environments, and virtual environments) guarantees (to differing degrees—at the hardware level, operating system level, Python environment level, and package environment level, respectively) that the relevant code will run similarly for different users. However, each of these systems either relies on additional software that can be complex or resource-intensive to install and use, or requires additional setup, creating potential barriers to both contributing to and taking advantage of open science resources. For example, what is typically "shared" with the end user in these cases is a set of instructions or configuration files for *building* the desired computing environment, rather than the environment itself.

We designed `Davos` to occupy a "sweet spot" within this space. `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 from the user, beyond what is needed to run a standard Jupyter notebook. And like separate dependency specification files that list packages for populating virtual environments or Python environments, `Davos` provides a convenient mechanism for fully specifying (and installing, as needed) a complete set of Python dependencies, including specific package versions, which are contained and isolated from the rest of the user's system.

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 [16]. The `smuggle` keyword provided by `Davos` is a play on Python's `import` keyword: whereas importing can load 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), the `Davos` package uses "onion comments" to precisely control how packages are smuggled into the Python workspace.

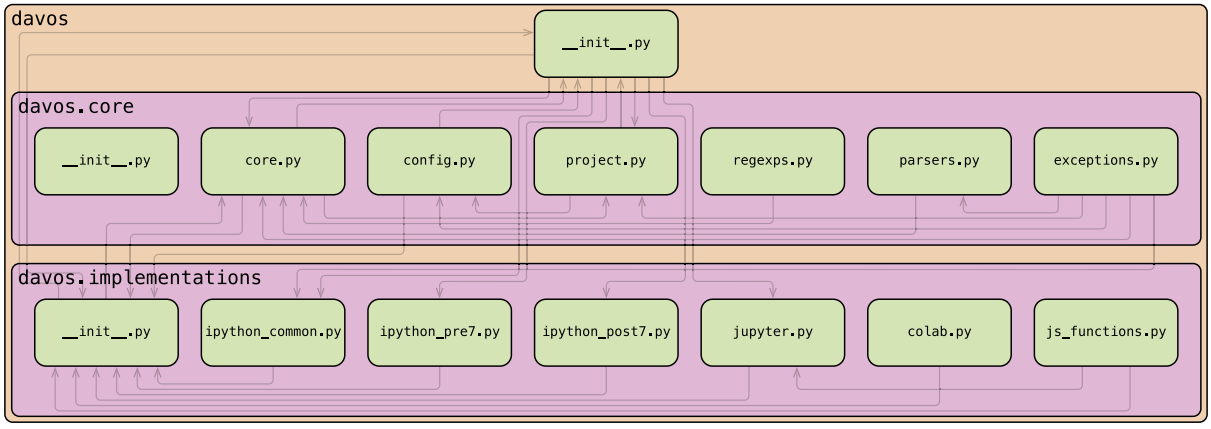


Fig. 2. Package structure. The `Davos` package comprises two interdependent subpackages. The `davos.core` subpackage includes modules for parsing `smuggle` statements and onion comments, installing and validating packages, isolating and managing installed packages, and configuring `Davos`'s behavior. The `davos.implementations` subpackage includes environment-specific modifications and features that are needed to support the core functionality across different notebook-based environments. Individual modules (i.e., `.py` files) are represented by lime rounded rectangles, and arrows denote dependencies (each arrow points to a module that imports objects defined in the module at the arrow's source).

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), a system for isolating dependencies of different projects (see Section 2.2.3), and a runtime interface for configuring `Davos`'s behavior (see Section 2.2.4). 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 [17] is used by the notebook kernel). To deal with this, environment-dependent components 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 is that it allows maintainers and users to extend `Davos` to support new, updated, or custom notebook variants by adding 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 can be used as a drop-in replacement for Python's built-in `import` keyword to load packages, modules, and other objects into the notebook's 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 user's Python environment, `Davos` will download and install it automatically, expose its contents to Python's `import` machinery, and load it into the notebook for immediate use.

Importantly, packages installed by `Davos` are made available for use in the notebook without affecting the user's Python environment or existing packages. By default, `smuggle` statements will install missing packages (and any missing dependencies of those packages) into a notebook-specific, virtual environment-like directory called a "project" (see Section 2.2.3). In turn, `smuggle` statements executed in a particular notebook will preferentially load packages from that notebook's project directory whenever they are available, rather than searching for them in the user's main Python environment. In this way, `smuggle` statements can be substituted for `import` statements to automatically ensure that all packages needed to run a notebook are installed and available at runtime each time the notebook is run, without risking interfering with dependencies of the user's other Python programs, or other `Davos`-enhanced notebooks.

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 [18], 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 (if necessary) without using or writing to the package cache
smuggle pandas as pd                 # pip: pandas --no-cache-dir

# pip-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 package if it 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 program'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 download, 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 [19] 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 shared notebooks from executing arbitrary code in a user's shell, this enables `Davos` to adjust its behavior based on how particular flags will affect the behavior of the installer program. For example, including `pip`'s `--no-input` flag will also temporarily enable `Davos`'s non-interactive mode (see Section 2.2.4). Similarly, if an onion comment contains either `-I`/`--ignore-installed`, `-U`/`--upgrade`, or `--force-reinstall`, `Davos` will install and load a new copy of the smuggled package without first checking for it locally:

```
# install and load hypertools v0.7 even if it already exists locally
smuggle hypertools as hyp        # pip: hypertools==0.7 —ignore-installed

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

Since the purpose of an onion comment is to describe how a smuggled package should be installed (if necessary) so that it can be loaded and used immediately, options that would normally cause the package not to be installed (such as `-h`/`--help` or `--dry-run`) are disallowed. Additionally, when using a `Davos` "project" to isolate smuggled packages (the default behavior; see Section 2.2.3), onion comments may not contain options that would change the package's installation location (such as `-t`/`--target`, `--root`, or `--prefix`). However, if the user disables project-based isolation and specifies `--target <dir>`, `Davos` will ensure that `<dir>` is included in the module search path (i.e., `sys.path`), prepending it if necessary, so the package can be loaded.

2.2.3. Projects

Standard approaches to installing packages from within a notebook can alter the local Python environment in potentially unexpected and undesired ways. For example, running a notebook that installs its dependencies via system shell commands (prefixed with "!") or IPython magic commands (prefixed with "%") may cause other existing packages in the user's environment to be replaced with alternate versions. This can lead to incompatibilities between installed packages, affect the behavior of the user's other scripts or notebooks, or even interfere with system applications.

To prevent `Davos`-enhanced notebooks from having unwanted side effects on the user's environment, any packages installed via `smuggle` statements are automatically isolated using a custom, virtual environment-like system called "projects". `Davos` projects are similar to standard Python virtual environments (e.g., created with the standard library's `venv` module or a third-party tool like `virtualenv` [7]) but with a few noteworthy differences that make them generally lighter-weight and simpler to use. Like a standard virtual environment, a `Davos` project consists of a directory (within a hidden `.davos` folder in the user's home directory) that houses third-party packages needed for a particular Python project, workflow, or task. However, unlike standard virtual environments, `Davos` projects do not need to be manually created, activated, or deactivated, and they function to *extend* the user's existing Python environment rather than replace it.

When `Davos` is imported into a notebook, a project directory for that notebook is automatically created (if it does not exist already). When `smuggle` statements within that notebook are executed, any packages (or specific versions of packages) that are not already available in the user's Python environment are installed into the notebook's project directory (along with any missing dependencies of those packages). During each `smuggle` statement's execution, `Davos` also temporarily prepends the notebook's project directory to the module search path so that these project-installed packages are visible when searching for smuggled packages locally, and prioritized over those in the user's main environment.

Thus, rather than constructing fully separate Python environments from scratch, `Davos` projects work by supplementing the user's runtime environment with any additional packages (or specific package versions) needed to satisfy the dependencies of their corresponding notebooks. In some cases, this might include every package smuggled into a notebook (e.g., if the notebook is run inside a freshly created, empty virtual environment). In other cases, the user's environment may already provide all required packages, and the notebook's project directory will go unused (in which case it will be deleted automatically when the notebook kernel is shut down). Regardless of the extent to which the existing environment is augmented, `Davos`'s project system ensures that all smuggled packages are installed locally and loaded successfully at runtime, while the contents of the user's Python environment are never altered.

Because `smuggle` statements in a given notebook are evaluated every time the notebook is run, this ensures that the notebook's requirements will remain satisfied even if the user's Python environment changes. For example, suppose a user has `NumPy` [20] v1.24.3 installed in their current Python environment and runs a `Davos`-enhanced notebook that smuggles `NumPy` with "`numpy==1.24.3`" specified in an onion comment (see Section 2.2.2). Since the user's existing version of the package satisfies this requirement, `Davos` will load it into the notebook's runtime environment. But if the user later upgrades their environment's `NumPy` version to v1.25.0 (perhaps as a result of installing a different package that depends on it) and subsequently re-runs this notebook, the local version will longer satisfy this requirement, so `Davos` will install `NumPy` v1.24.3 into the notebook's project directory and load that version instead. From then on, any further changes to the user's `Numpy` installation would have no effect on `Davos`'s behavior in this particular notebook, as a satisfactory version now exists in its project directory. (If the version specified in the onion comment were changed, `Davos` would update the version installed in the project directory accordingly.) For efficiency, `Davos` projects will generally not duplicate dependencies already satisfied by the user's Python environment. However, if desired, adding `pip`'s `--ignore-installed` flag to an onion comment in the notebook will cause `Davos` to install the smuggled package into the project directory whether or not it already exists locally.

By default, each Davos-enhanced notebook will create and use its own notebook-specific project named for the absolute path to the notebook file. However, before smuggling its required packages, a notebook may be set to instead use an arbitrarily named, notebook-agnostic project by assigning any (non-empty) string to `davos.project` (see Section 2.2.4). This provides a convenient way for multiple related notebooks that share a common set of requirements to use the same Davos project, by setting `davos.project` to the same string in each one. It is also possible (though typically not recommended) to disable Davos's project system and instead install smuggled packages directly into the user's Python environment by setting `davos.project` to `None`.

When accessed (unless its value has been set to `None`), `davos.project` will evaluate to a `Project` object that represents the project used by the current notebook (strings assigned to `davos.project` are converted to `Projects` internally). This object supports methods for interacting with the current project, including locating its directory within the file system, listing all installed packages' names and versions, changing the project's name, and deleting its contents. `Project` instances can also be created and managed programmatically, and Davos provides additional utilities for viewing and working with all existing projects (see Sections 2.2.4 and 2.2.5).

2.2.4. *Configuring and querying Davos*

After importing Davos into a notebook, the top-level `davos` module exposes a set of attributes whose values determine various aspects of Davos's behavior. The majority of these are writeable options that can be modified to customize how, where, and when Davos installs smuggled packages (see Section 3 for an illustrative example). 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, `davos.active` is set to `True` (see Section 2.3 for implementation details and additional information).
- `.auto_rerun`: This attribute controls how Davos behaves when attempting to `smuggle` a new version of a package that was previously loaded (via an `import` or `smuggle` statement) and cannot be reloaded. This can happen if the package includes extension modules that dynamically link C or C++ objects to the Python interpreter, and the code that generates those objects was changed between the previously loaded and to-be-smuggled versions. If this attribute is set to `True`, Davos will automatically restart the notebook kernel and re-run all code up to (and including) the current `smuggle` statement. If set to `False` (the default), Davos will instead issue a warning, pause execution, and prompt the user to either restart and re-run the notebook, or continue running with the previously loaded package version until the next time the kernel is restarted manually. Note that, as of this writing, setting `davos.auto_rerun` to `True` is not supported in Google Colaboratory notebooks.
- `.confirm_install`: If set to `True` (default: `False`), Davos will require user confirmation before installing a smuggled package that is not already available locally. This is primarily useful if the user has disabled Davos's "project" system for isolating smuggled packages (see Section 2.2.3) but still wants to carefully control what packages are installed into their main Python 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 set to `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 user's Python environment and falls back to `<sys.executable> -m pip` if no executable can be found.
- `.project`: This attribute's value is a `Project` instance representing the Davos project associated with the current notebook. As described in Section 2.2.3, Davos projects serve to isolate packages installed by `smuggle` statements from the user's main Python environment, and the `Project` class provides an interface for inspecting and managing projects at runtime. This attribute's default value is a notebook-specific project named for the absolute path to the notebook file. To change the project used in the current notebook (e.g., in order to use the same project in multiple related notebooks), this attribute may be assigned a different `Project` instance or, for simplicity, the name of the desired project as a string or `pathlib.Path` (either of which will be converted to a `Project` on assignment). Alternatively, setting `davos.project` to `None` will disable project-based isolation for the current notebook and cause Davos to install any missing packages directly into the main Python environment. This attribute can be reset to its default value using the top-level `use_default_project()` function (see Section 2.2.5). For more information about Davos projects, see Section 2.2.3.
- `.suppress_stdout`: If this attribute is set to `True` (default: `False`), Davos suppresses printed (console) outputs from both itself and the installer program. This can be useful when smuggling packages that require installing many dependencies and/or generate extensive output when built from source distributions. Note that if this option is enabled and the installer program throws an error, both its stdout and stderr streams will still be displayed alongside the Python traceback to allow for debugging.

The attributes above can be modified directly or via the `davos.configure()` function, which allows setting multiple options simultaneously (see Section 2.2.5 for more information or Section 3 for example usage). In addition to these writeable options, the top-level `davos` module also provides several read-only attributes that can be displayed in the notebook or checked programmatically at runtime, and contain potentially useful information about the notebook environment or Davos's internal state:

- `.all_projects`: This attribute contains a list of all Davos projects that exist on the user's local system (see Section 2.2.3 for more information about Davos projects). Each item in this list is either a `Project` or `AbstractProject` instance. `AbstractProjects` represent notebook-specific projects whose associated notebooks no longer exist. They support the same functionality as `Project` objects (including methods for inspecting, renaming, and deleting them) and serve primarily to help users identify and clean up extraneous projects left behind after deleting Davos-enhanced notebooks (e.g., see Section 2.2.5).
- `.environment`: This attribute's value is a string denoting the set of environment-dependent "helper functions" used by Davos in the current notebook. As described in Section 2.1, Davos internally chooses between interchangeable implementations of certain core features based on various properties of the notebook's frontend and IPython kernel. As of this writing, three unique combinations of helper functions are required to support existing notebook environments, ergo this attribute has three possible values: `"IPython<7.0"`, `"IPython>=7.0"`, or `"Colaboratory"`. However, this attribute could take on additional values in the future as new notebook interfaces are created and IPython's internals are updated, and as additional versions of helper functions are added to Davos to support them.

- `.ipython_shell`: This attribute contains the global IPython `InteractiveShell` instance underlying the notebook kernel session.
- `.smuggled`: This attribute's value is a Python dictionary that functions as a cache of `smuggle` statements executed during the current notebook kernel session. The dictionary's keys are names of smuggled packages, and its values are arguments passed to the installer program via onion comments. Entries appear in order of the `smuggle` statements' execution.

The current values of all `davos` attributes may be viewed at once within a notebook by printing the `davos.config` object.

2.2.5. Other top-level `Davos` functions

The `Davos` package also provides a handful of functions available in the top-level `davos` namespace. Some of these functions serve primarily as conveniences, while others provide additional functionality:

- `configure(**kwargs)`: This function provides an alternate way of assigning values to the writeable attributes listed in Section 2.2.4 and can be used to configure multiple options at once (see Section 3 for example usage). The function accepts attribute names as keyword-only arguments to which their desired values are passed. If any of the options passed are incompatible (e.g., both `confirm_install=True` and `non-interactive=True` are passed) or assignment to any of the specified attributes fails for any reason, none of the given options will be modified.
- `get_project(project_name, create=False)`: This function can be passed the name of a `Davos` project (`project_name`) to get the `Project` or `AbstractProject` instance representing it. The optional `create` argument determines the function's behavior when no project with the given name exists: if `create=False` (the default), the function will return `None`; if `create=True`, a project with the given name will be created and returned.
- `prune_projects(yes=False)`: This function allows users to quickly "clean up" their local `Davos` projects by deleting notebook-specific projects whose corresponding notebooks no longer exist (i.e., `AbstractProjects`). As with standard virtual environments, periodically removing unused project directories can be useful for reclaiming disk space from dependencies of code that is no longer in use. By default, this function will interactively display a list of all unused projects and allow the user to choose whether or not to delete each one. Alternatively, passing `yes=True` will immediately remove all unused projects without prompting for confirmation. Note that if `Davos`'s non-interactive mode is enabled (see Section 2.2.4), `yes=True` must be explicitly passed, otherwise the function will raise an exception. This serves as a safeguard against accidentally deleting projects, since non-interactive mode disables all user input and confirmation. Also note that this function will not delete notebook-agnostic projects (i.e., manually created projects whose names are not notebook file paths), as they are not linked to specific notebooks whose existence determines whether or not they are still needed. These (and any) projects may be deleted individually by calling their `Project` objects' `.remove()` method.
- `require_python(version_spec, warn=False, extra_msg=None, prereleases=None)`: Through `smuggle` statements and onion comments, `Davos` can automatically ensure that all Python packages needed to run a notebook are installed, and that the same versions of those packages are used no matter when or by whom the notebook is run. However, because `Davos` operates at runtime, one thing it cannot do automatically is install and switch to a specific version of Python itself. Distributing shared code along with a precise Python version for running it requires a heavier-weight solution, such as a Conda environment or Docker container (see Fig. 1). Yet a `Davos`-enhanced notebook may still `smuggle` certain packages that depend on users having a particular Python version or range of versions (e.g., even just within the standard library, the `dataclass` module was first added in Python 3.7 [21] and at least 19 modules are slated for removal in Python 3.13 [22]). The `davos.require_python()` function can be added to the top of a `Davos`-enhanced notebook to communicate to users that the notebook's code should be run with a specific or constrained Python version (see Section 3 for example usage). The function may be passed a version identifier (e.g., `"3.10.5"`) or any valid version specifier [4] (e.g., `"~=3.11"`, `">=3.9;<3.12"`, etc.) and will raise an exception if the user's Python version is incompatible. Alternatively, a "soft" or suggested constraint can be imposed by passing `warn=True` to issue a warning rather than raise an error. Additional information can be added to the default error/warning message (e.g., the specific reason for this requirement) via the `extra_msg` argument, and the optional `prereleases` argument can be used to explicitly allow (`True`) or disallow (`False`) pre-release versions (by default, the policy is determined by the value of `version_spec`).
- `use_default_project()`: By default, each `Davos`-enhanced notebook will create and use a notebook-specific project named based on its absolute path. If a user manually changes the project used by the current notebook (i.e., by setting the value of the `davos.project` attribute; see Section 2.2.4), this function can be called to switch back to using the notebook's default project and reset `davos.project` to its default value. See Section 2.2.3 for more information about `Davos` projects and Section 3 for an illustrative example.

2.3. Implementation details

Although `Davos` is designed to *appear* to add a new keyword to Python's vocabulary, this illusion is actually created through several "hacks" that make use of the notebook's IPython backend for processing and executing users' code. Specifically, when `Davos` is first imported, or when it is activated after having been 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 `!`-prefixed shell commands and `%`-prefixed "magic" commands. `Davos` uses this same process to invisibly 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 Fig. 3. Because the `smuggle()` function is defined in the notebook namespace, it is

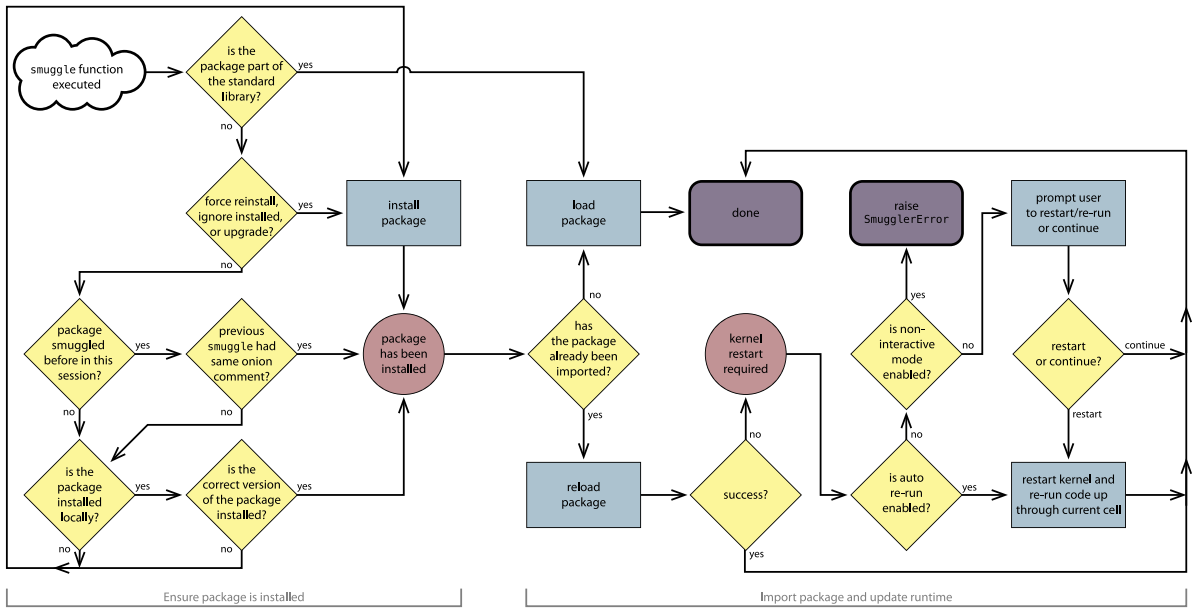


Fig. 3. `smuggle()` function algorithm. At a high level, the `smuggle()` function may be conceptualized as following two basic steps. First (left), `Davos` ensures that the correct version of the desired package is available locally, installing it automatically (into the notebook's project directory) if needed. Second (right), `Davos` loads the package into the notebook and updates the current runtime environment.

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.

3. Illustrative example

The example code throughout Section 2.2.2 illustrates a typical use case that we envision for `Davos`: a series of smuggle statements and onion comments with version specifiers or other options collectively describes and automatically constructs a reproducible environment for running the code that follows it. When added to the top of a Jupyter notebook, this allows researchers to bundle their code and its dependencies into a single file that can be easily shared and run without any additional tools or setup, automatically installs its required packages at runtime, isolates them from the user's main Python environment, and ensures their versions do not change unexpectedly over time. In this section, we have contrived a more complex scenario to highlight some of `Davos`'s more advanced features, and illustrate how they may be used to handle certain challenges that can arise when writing, running, and sharing reproducible scientific code.

Across different versions of a given package, various modules, functions, and other objects may be updated, removed, renamed, or otherwise altered. In addition to changing the behaviors of active computations, these changes can render saved objects created using one version of a package incompatible with other versions of the same package. For example, the popular `pandas` [23] library originally included the `Panel` data structure for storing 3-dimensional arrays. In version 0.20.0, however, the `Panel` class was deprecated, and in version 0.25.0, it was removed entirely. Suppose a user had a dataset stored in a `Panel` object (created using an older version of `pandas`) and had saved it to their disk (e.g., for later reuse or to share with other users) by serializing the `Panel` with Python's `pickle` protocol. The `pickle` protocol is a popular built-in method of persisting data in Python that allows users to save, share, and load arbitrary objects. However, in order to successfully "unpickle" (i.e., load and restore) a "pickled" (i.e., previously saved) object, that object's class must be defined in and importable from the same module as it was when the object was originally saved. Thus, because of the `Panel` class's removal, the user's dataset could not be read by any version of `pandas` from 0.25.0 onward. These incompatibilities are also not limited solely to traditional forms of data. For example, saved model states and other objects may reference modules, functions, attributes, classes, or other objects that may not be identical (or even present) across all versions of their associated packages.

The example provided in Fig. 4 demonstrates how `Davos` can be used to circumvent these incompatibilities by temporarily switching between different versions of the same package within a single runtime. The example shows how a dataset and model that require now-incompatible components of the `pandas` and `scikit-learn` [24] libraries can be loaded in (using older versions of each package) and used alongside more recent versions of each package that provide new and improved functionality. When included at the top of a Jupyter notebook, the code in Fig. 4 ensures that these objects will be loaded successfully and analyzed using the same set of package versions no matter when or by whom the notebook is run.

After installing and importing `Davos` (lines 1–2), we first use the `davos.require_python()` function to constrain the Python version used to run the notebook (see Section 2.2.5). As described above, the example code in Fig. 4 loads two different versions of the `pandas` library: first, an older version needed to access a dataset saved in an outmoded format, then a newer one to use throughout the remainder of the notebook. We therefore want to make sure upfront (in line 6) that the notebook's Python version falls within the range of versions that both of these two versions of `pandas` support. If it does not, the function in line 6 will raise an error that includes a message to this effect (lines 4–5).

```
1  %pip install davos
2  import davos
3
4  extra_msg = ("pandas<0.25.0 is needed to load the dataset and requires Python<3.8. "
5               "pandas==1.3.5 is used to run the analyses and requires Python>=3.7.1.")
6  davos.require_python(">=3.7.1,<3.8", extra_msg=extra_msg)
```

Next, in lines 8–9, we `smuggle` two utilities for interacting with local files in the code below. The `smuggle` statement in line 8 loads the `is_file()` function from the Python standard library's `os.path`

```
%pip install davos
import davos

extra_msg = ("pandas<0.25.0 is needed to load the dataset and requires Python<3.8. "
             "pandas==1.3.5 is used to run the analyses and requires Python>=3.7.1.")
davos.require_python(">=3.7.1,<3.8", extra_msg=extra_msg)

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

davos.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.project = None
kernel_env_pip = davos.pip_executable
server_env_pip = !command -v pip
davos.pip_executable = server_env_pip[0]
smuggle widgetsnbextension as _
davos.use_default_project()
davos.pip_executable = kernel_env_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
```

Fig. 4. Example use case for `Davos`. Snippets from this example are also excerpted in the main text of Section 3.

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 (since there is no chance the module will need to be installed). Line 9 then loads the `joblib` package [25], installing it into the notebook's project directory if necessary. Since `joblib`'s I/O interface has historically remained stable and backwards-compatible across releases, requiring a particular exact version would likely be unnecessarily restrictive. However, it is possible a *future* release could introduce some breaking change. The onion comment in line 9 helps ensure that the analysis notebook will continue to run properly in the future by limiting allowable versions to those already released when the code was written:

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

It is worth noting, however, that beyond illustrative purposes, the benefit of specifying only a maximum version for `joblib` rather than an exact version is relatively minor. The main advantage to relaxing a version constraint in an onion comment (when a package's behavior does not differ meaningfully between versions) is that doing so increases the likelihood that a satisfactory version will already be available in the user's Python environment, and therefore `Davos` will not need to install a new copy in the notebook's project directory. For large packages, this can be a worthwhile consideration; however `joblib` is very lightweight—less than 0.5 MB pre-built, with no required dependencies. Thus a more conservative approach that guarantees an exact version is used would also be reasonable in this case.

Line 11 then enables `Davos`'s `auto_rerun` option (see Section 2.2.4) before smuggling the next two packages: `NumPy` and `pandas`. Because these packages rely heavily on custom C data types, loading the particular versions specified in their onion comments may require restarting the notebook kernel if different versions were previously imported during the same interpreter session—including internally by other packages. Enabling `auto_rerun` allows `Davos` to handle kernel restarts automatically and continue running the code seamlessly without user intervention.

```
11  davos.auto_rerun = True
12  smuggle numpy as np               # pip: numpy==1.21.6
```

In the case of `NumPy`, whether or not a kernel restart is necessary will depend on the user's existing Python environment. The `joblib` package has an optional dependency on `NumPy` for memoizing and parallelizing array operations, and will `import numpy` internally to enable these features if the package is available. If the user already has `NumPy` installed in their Python environment when `joblib` is smuggled in line 9, their installed version is different from the one specified in the onion comment on line 12, and there were changes made to `NumPy`'s C extensions between those two versions, then `Davos` will automatically restart the kernel and re-run the lines above. The newly smuggled version would then be used both in the notebook itself and by `joblib` internally.

The primary reason for enabling the `auto_rerun` option, however, is to manage the installation of `pandas` in the next lines:

```
14  if not is_file("~/datasets/data-new.csv"):
15      smuggle pandas as pd          # pip: pandas<0.25.0
16      tmp_data = pd.read_pickle("~/datasets/data-old.pkl")
17      tmp_data.to_frame().to_csv("~/datasets/data-new.csv")
18
19  smuggle pandas as pd              # pip: pandas==1.3.5
```

If we suppose that the "`data-old.pkl`" file contains a dataset stored in a pickled `Panel` object, then we must use a version of `pandas` prior to v0.25.0 (i.e., the version in which the `Panel` class was removed) to be able to read it. Line 15 ensures that a sufficiently old version of `pandas` will be imported, enabling the data to be successfully loaded in line 16 and (in line 17) written to a CSV file, which can be read by any `pandas` version.

Newer versions of `pandas` have brought substantial improvements including performance enhancements, bug fixes, and additional functionality. Although the original dataset had to be read in using an older version of the package, we can take advantage of these more recent updates by smuggling `pandas` a second time in line 19 (whose onion comment specifies that version 1.3.5 should be installed and loaded). Since a different `pandas` version has already been loaded by the Python interpreter (line 15) and there have been substantial changes to the library (including its extension modules) between that version and v1.3.5, the notebook kernel must be restarted in order to fully unload the old version in favor of the new one. When `Davos` automatically does so and re-runs the code above, having now converted the dataset to a CSV file means the old version does not need to be reinstalled (line 14).

Next, line 21 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 22–23 `smuggle TensorFlow` [26], a powerful end-to-end platform for building and working with machine learning models, and `UMAP` [27], a package that implements a family of related manifold learning techniques. The onion comment in line 23 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. If many of these are not already installed in the user's environment, lines 22–23 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 while installing these packages and their many dependencies to prevent other potentially important outputs from being buried.

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

After reverting these two options (line 24) to their default values, we next `smuggle` specific versions of three plotting packages: `Matplotlib` [28], `seaborn` [29], and `Quail` [30] (lines 26–28). Because the first two are requirements of `UMAP`'s optional "plot" feature, they will have already been installed (if necessary) by line 23, though possibly as different versions than those specified in the onion comments on lines 26 and 28. If the installed and specified versions are the same, these `smuggle` statements will function like standard `import` statements to load the packages into the notebook's namespace. If they differ, `Davos` will download the requested versions in place of the installed versions, ensuring that they are used both in the notebook itself and by `UMAP` internally.

```
24  davos.configure(suppress_stdout=False, noninteractive=False)
25
26  smuggle matplotlib.pyplot as plt  # pip: matplotlib==3.5.3
27  smuggle seaborn as sns            # pip: seaborn==0.12.1
28  smuggle quail                     # pip: git+https://github.com/myfork/quail@6c847a4
```

The onion comment in line 28 specifies that `Quail` should be installed from a fork of its GitHub repository (`myfork`), in its state as of a specific commit (`6c847a4`). This ability to load packages directly from remote (or local) Git repositories can enable developers to more easily use forked or customized versions of other packages in their code, even if those versions have not been officially released. Targeting specific VCS references (e.g., commits, tags, etc.) can also provide even finer-grained control over smuggled package versions than is possible with traditional version specifiers.

In lines 30–37, we demonstrate another aspect of `Davos`'s functionality that supports more advanced installation scenarios. The `ipywidgets` [31] package (also known as Jupyter Widgets) provides a Python API for creating interactive JavaScript widgets within a notebook. It depends on the `widgetsnbextension` package, which provides the JavaScript machinery needed by the notebook frontend to display these widgets. A complication is that `ipywidgets` must be installed in a location that is accessible from the IPython kernel (i.e., the Python runtime within the notebook itself), while `widgetsnbextension` must be installed in the environment that houses the Jupyter notebook server (a separate Python runtime that serves and manages the notebook frontend client). In many basic setups, the IPython kernel and notebook server exist in the same environment.

However, a common "advanced" approach entails running the notebook server from a base environment, with additional environments each providing their own separate, interchangeable IPython kernels.

Lines 30–37 account for both of these possibilities programmatically:

```
30  davos.project = None
31  kernel_env_pip = davos.pip_executable
32  server_env_pip = !command -v pip
33  davos.pip_executable = server_env_pip[0]
34  smuggle widgetsnbextension as _
35  davos.use_default_project()
36  davos.pip_executable = kernel_env_pip
37  smuggle ipywidgets                  # pip: ipywidgets==7.6.5
38
39  from tqdm.notebook smuggle tqdm     # pip: tqdm==4.62.3
```

First, in line 30, we set the `davos.project` attribute to `None` to temporarily allow installing smuggled packages outside of the notebook's project directory. As noted in Section 2.2.3, this is typically discouraged, as doing so can risk interfering with the user's Python environment if existing package versions are overwritten. In this particular case, however, a combination of factors make this relatively safe and inconsequential. First, the package we need to install directly into the notebook server environment (`widgetsnbextension`) is smuggled without an accompanying onion comment (line 34), meaning that `Davos` will not replace any version the user may already have installed. Second, the package has no dependencies of its own, so if `Davos` does install it, no other packages will be installed or updated as a side effect. Third, the package itself provides no functionality outside of rendering Jupyter widgets, so its presence would not alter any other code's expected behavior.

Next, in lines 31–33, we change the `pip` executable `Davos` uses to install smuggled packages (see Section 2.2.4), storing the default executable's path in a variable before doing so. When `Davos`'s project system is disabled, using a `pip` executable from a particular Python environment will cause smuggled packages to be installed into (and subsequently loaded from) that environment. The default `pip_executable` will install packages into the environment used to run the IPython kernel. Here, the new value assigned to `davos.pip_executable` in line 33 is the output of running "`command -v pip`" as a `!`-prefixed IPython system shell command in line 32 ("`command -v`" outputs the path to an executable, similar to "`which`" but more portable). IPython system shell command are always executed in the notebook server's environment—which may or may not be different from the kernel's environment—so this command's output will be the path to the server environment's `pip` executable.

After smuggling the `widgetsnbextension` package in line 34, we use the `davos.use_default_project()` function in line 35 to revert to installing package into the notebook's project directory, restore the default value of `davos.pip_executable` in line 36, and `smuggle` the specified version of `ipywidgets` in line 37. With these two packages now installed and imported, line 39 smuggles `tqdm` [32], which displays progress bars to provide status updates for running code. In Jupyter notebooks, the `tqdm.notebook` module can be imported to enable more 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 first ensure that both `ipywidgets` and `widgetsnbextension` were available.

Next, we load in the reformatted dataset (line 41) and pre-trained model (line 43) 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 pre-trained models in succession. First, a trained `CountVectorizer` instance converts text data to an array of word counts. The word counts are then passed to a topic model [33] using a pre-trained `LatentDirichletAllocation` instance.

```
41  data = pd.read_csv("~/datasets/data-new.csv", index_col=[0, 1])
42  smuggle sklearn                     # pip: scikit-learn<0.22.0
43  transformer = joblib.load("~/models/text-transformer.joblib")
44  smuggle sklearn                     # pip: scikit-learn==1.1.3
```

Let us suppose that the `Pipeline` object had been saved by its original creator using the `joblib` package, as `scikit-learn`'s documentation recommends [34]. 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 (and previous versions) of `scikit-learn`, the `LatentDirichletAllocation` class was defined in the `sklearn.decomposition.online_lda` module. However, in version 0.22.0, that module was renamed to "`_online_lda`" and in version 0.22.1, it was again renamed to "`_lda`".

In order to successfully load the model that includes the pre-trained `LatentDirichletAllocation` instance, in line 42, we first `smuggle` 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 43), we upgrade to a newer version of `scikit-learn` in line 44. Taken together, the code in Fig. 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 manipulate the data and model output using the latest approaches and implementations.

4. Impact

We designed `Davos` for use in research settings, where code for numerous different tasks—from processing data, to running statistical analyses, to generating figures and tables for publication—is frequently shared between collaborators while working on a project, and eventually with the broader scientific community and general public upon its completion. In these contexts, ensuring that shared code yields consistent, reproducible outputs across users and over time is critical, yet the tools available to researchers for doing so can be complex to set up and challenging to properly use. This has the dual effect of discouraging scientists from sharing their code in a reproducible way (or at all), and making it significantly harder (or impossible) for others to successfully reproduce their results, adopt or extend their methods, and contribute to or build upon their work. Ultimately, the need to install and master additional tools in order to share and run reproducible code can impede progress both on the individual level and on the broader scientific level.

While the `Davos` package by no means offers a universal solution to this problem, it can, in many cases, provide an effective yet more accessible alternative to existing systems for sharing reproducible Python code, such as instructions and specification files for building and populating virtual environments, Python environments, containers, and virtual machines (Fig. 1). For researchers, this can lower barriers to collaborating with peers and contributing to publicly available open science resources. And by eliminating most of the setup costs of reconstructing the original researchers' computing environment(s), `Davos` also lowers barriers to entry for members of the scientific community and the public who seek to run shared code.

Among common systems for sharing reproducible Python code (see Fig. 1), `Davos` is most comparable to instructions and specification files for building and populating virtual environments in that it provides a lightweight mechanism for specifying and sharing a complete set of packages (and specific package versions) required by a particular project, and installing those packages into an isolated directory on the user's file system. However, when used in conjunction with Jupyter notebooks, `Davos` offers a number of advantages over standard virtual environments that make it both easier to use and more effective at ensuring reproducibility of shared code.

First, `smuggle` statements and onion comments enable researchers to specify their notebooks' dependencies directly within the code that requires them (see Sections 2.2.1 and 2.2.2). This eliminates the need to either manually create and maintain separate configuration files or use an additional tool to generate them, and to then distribute these files alongside their primary code base for users to download.

Second, `Davos` automatically checks for, installs, isolates, and imports any required packages at runtime. This allows users to download and immediately run a `Davos`-enhanced notebook without any prerequisite setup (beyond that needed to run a standard Jupyter notebook). By contrast, before running a notebook whose dependencies are managed via a virtual environment, the user would first need to run a series of shell commands to manually create a new environment, populate it with packages from the researcher's configuration file (which the user must also have obtained), and then either use the `ipykernel` package to register the environment as a Jupyter kernel, or activate and deactivate the environment before and after (respectively) each time the notebook was launched. Beyond reducing this initial setup cost, `Davos`'s runtime-based approach to dependency management affords a second important benefit. While creating and configuring a virtual environment ensures that a specific set of packages is initially installed, it does not guarantee that they will *remain* installed after that point. For example, a researcher who creates a virtual environment in which to run a set of data analyses—or a different user who later recreates that environment to reproduce them—might at some point install an additional package into the environment after its initial setup (e.g., to implement a new analysis idea). Depending on the requirements of this new package, this could cause one or more initially installed packages to be upgraded or downgraded to a different version. If the individual does not happen to notice this change when it occurs, differences between those packages' expected and installed versions may introduce bugs into previously written code or subtly alter its output when it is next run. Because `smuggle` statements and onion comments are evaluated every time a `Davos`-enhanced notebook is run, they function to ensure that the notebook's dependencies are always satisfied and that any such inadvertent changes would be automatically caught and corrected.

Third, running a shared notebook that uses `Davos` to manage its dependencies often requires less (but never more) space on the user's system than running an identical notebook inside a virtual environment. While typical virtual environment directories will contain *all* requirements listed in their configuration files, `Davos`'s isolated project directories (by default; see Section 2.2.3) contain only those not already available in the user's existing Python environment. This is another feature made possible by `Davos`'s runtime-based dependency management system. `Davos` can safely draw from packages in the user's main environment to satisfy a notebook's dependencies, when possible, because if those packages were to be updated or removed at any point such that they no longer met the notebook's requirements, appropriate versions would be installed into the project directory the next time the notebook was executed.

Finally, neither specification files used to install packages into virtual environments nor `smuggle` statements and onion comments included `Davos`-enhanced notebooks can be used to automatically install a specific version of Python with which to run shared code. However, `Davos`'s `require_python()` function provides a simple mechanism for indicating a required or constrained Python version and alerting the user at runtime (by raising an exception) if their Python version is incompatible. We note that a similar constraint can be imposed by setting the "`requires-python`" attribute in a `pyproject.toml` file, though creating a virtual environment from a `pyproject.toml` would require installing an additional third-party tool (such as [8]). In terms of `Davos`'s ability to ensure that shared code can be executed reproducibly by other users, this falls short of the capabilities of more complex tools that can provide complete copies of Python (Fig. 1). However, the functionality `Davos` provides over what is possible with a standard virtual environment is to remove the *expectation* that running a particular notebook will reproduce an expected outcome in situations where this is either impossible or not guaranteed. With this understanding, a user may choose to install a compatible Python version through some other means or elect to still run the code, but will not be surprised by a potential failure to execute successfully or output an expected result.

Beyond research applications, `Davos` is also useful in pedagogical settings. For example, in programming courses, instructors and students may use the `Davos` package 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 prior and ongoing research studies [35–37], `Davos` is being used by both students and instructors in programming and research methods courses such as Storytelling with Data [38] (an open course on data science, visualization, and communication), Laboratory in Psychological Science [39] (an open course on experimental and statistical methods for psychology research), and the Methods in Neuroscience at Dartmouth (MIND) Computational Summer School [40] (a week-long intensive course on computational neuroscience methods) to simplify distributing lessons and submitting assignments, as well as in online demos such as `abstract2paper` [41] (an example application of GPT-Neo [42,43]) to share ready-to-run code that installs dependencies automatically. The 2023 offering of Neuromatch Academy [44] also included an "experimental" module that uses `Davos` to manage dependencies related to a large language model-based tutor [45].

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 and Fig. 4). 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., [28,46]), this approach also has the potential to be exploited for more nefarious purposes. For example, a malicious user could design a Python package 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. However, there have been early-stage discussions of providing this sort of syntactic customizability as a core feature of the Python language itself, including a draft proposal [47]. 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.

4.1. Pitfalls and limitations

While Davos enables developers to conveniently specify all project dependencies, there are some edge cases and limitations that are worth considering. First, prior studies on reproducibility of Jupyter notebooks [e.g., 5] identified a key challenge in the fact that, unlike Python scripts, notebook cells may be manually executed in an arbitrary order, and therefore potentially in a different order than they were executed by the notebook's original author. This can result in situations where, for example, a cell's execution fails because its code calls a function that has not yet been defined, or accesses a variable that refers to a different object than is expected at that point in the notebook. In theory, using Davos to smuggle multiple versions of the same package in different cells of a notebook could exacerbate this issue if a user executed those cells out of their intended order, such that their currently imported version of a core dependency was different from what a particular cell expected or required. Therefore, an important consideration when using Davos to facilitate complex, multi-package-version runtimes in this way is that executing notebook cells in order is perhaps even more important than it would be in a standard (i.e., non-Davos-enhanced) notebook. While (as noted in Sections 3 and 4) we consider this an "advanced feature" of Davos rather than typical usage, we propose a relatively simple set of "best practices" that substantially mitigate the risk of creating ambiguous states within a notebook. First, any Davos-enhanced notebook (or simply any notebook) that is intended to be run by more than one individual should be organized with its code cells in their intended execution order from top to bottom. If an edge case arises in which this is not possible, the intended order should be clearly indicated in code comments and/or markdown cells. Second, when smuggling multiple different versions of a package within a notebook, one version of the package may be designated the "main" version, and any others designated as "alternate" versions. The main version should be the primary version used throughout the notebook, while alternates are those temporarily required for a specific task or functionality. For example, in Fig. 4, `pandas` v1.3.5 and `scikit-learn` v1.1.3 are the main versions of their respective packages as they are used throughout the remainder of the code once they are loaded. Meanwhile `pandas<0.25.0` and `scikit-learn<0.22.0` are alternate versions because they are temporarily smuggled for the specific purpose of loading an outmoded dataset and model and then immediately replaced with main versions after their use is complete. Any time an alternate package version is needed, the `smuggle` statement used to install and load it, the operations it is required to perform, and a second `smuggle` to (re-)install and load the main package version should all be contained within a single notebook cell. This ensures that (barring other unrelated errors in the cell's execution) the main version will always be installed and imported when any given notebook cell is run. In other words, in Fig. 4, lines 14–19 should be run within a single cell, and lines 42–44 should also be run in a single cell.

A second limitation of Davos relates to how packages are installed and managed. As of this writing, Davos can install packages using `pip`, but not other standard Python package management systems such as `conda` [9]. Therefore packages that are not installable via `pip` are currently unsupported by Davos. We anticipate adding support for other package management systems, including `conda`, in a future release. Because Davos relies on `pip` to install packages, it is also subject to the same limitations as `pip` itself. For example, `pip`-installing a package that depends on a previously `smuggled` package may result in the previously smuggled package being upgraded or downgraded to a different version. Whereas lockfiles, or lockfile-based systems like `Poetry` [8], place stronger guarantees that each package will have a stable version, we have opted for a more flexible (but, consequently, less deterministic) implementation for Davos. This enables us to support more advanced use cases, such as those described in Section 3, but at the cost of managing potential conflicts between `smuggled` packages.

A third limitation of Davos is that it cannot be used to manage projects that depend on non-Python software. For example, system software or libraries from other languages (e.g., in a mixed Python and R notebook), cannot be `smuggled` by Davos. A notebook that utilizes or depends on non-Python software would therefore need to use existing non-Davos approaches to managing those requirements.

Davos's "projects" system (Section 2.2.3) provides a safe way of managing project dependencies without interfering with the user's Python environment. By default, each Davos-enhanced notebook creates and uses its own notebook-specific project directory, which is named based on the notebook's absolute path. However, programmatically determining the path to the currently running notebook may not be possible in some environments. For example, Davos queries the Jupyter Server API to determine the notebook's name, but some non-browser applications may implement mechanisms for communicating with the IPython kernel without starting a Jupyter server. As of this writing, Davos fully supports most common notebook environments, including classic Jupyter Notebooks, JupyterLab, Google Colaboratory, Binder, Kaggle Notebooks, JetBrains IDEs (PyCharm, DataSpell, etc.), and Visual Studio Code, among others. However, in cases where Davos fails to determine the current notebook's path, it will issue a warning and fall back to using a generic project named "davos-fallback". This project is shared across all such occurrences and exists to ensure that even if some component of Davos's project system fails, smuggling packages will still not affect the user's main Python environment. If this occurs, the user can also manually set Davos to use the "normal" default project for the current by setting `davos.project` to its absolute or relative path.

5. Conclusions

The Davos package supports reproducible research by providing a novel, lightweight system for sharing notebook-based code. It expands on Python's "batteries included" philosophy to enable running shared notebooks with no setup required, and defines a simple, self-documenting format for specifying dependencies in keeping with the "literate programming" paradigm that Jupyter notebooks support. We designed Davos to fill a niche we believe will help facilitate contributing to and engaging with open science resources. But perhaps the most exciting uses of the Davos package are those that we have *not* yet considered or imagined. We hope that the research and scientific Python communities will find Davos to provide a convenient means of managing project dependencies to facilitate code sharing and collaboration. We also hope that some of the more advanced applications of our package might lead to new insights or discoveries.

Funding

Our work was supported in part by National Science Foundation, United States of America 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.

CRediT authorship contribution statement

Paxton C. Fitzpatrick: Conceptualization, Methodology, Software, Validation, Writing – original draft, Visualization. **Jeremy R. Manning:** Conceptualization, Resources, Validation, Writing – review & editing, Visualization, Supervision, Funding acquisition.

Declaration of competing interest

The authors declare that they have no known competing financial interests or personal relationships that could have appeared to influence the work reported in this paper.

Data availability

No data was used for the research described in the article.

Acknowledgments

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 package in several assignments, and the students of the Methods in Neuroscience at Dartmouth (MIND) Computational Summer School (2023 offering) who used our package during several workshops and tutorials.

References

[1] van Rossum G. Python reference manual, vol. 111. Centrum voor Wiskunde en Informatica Amsterdam; 1995.
[2] Python Software Foundation. The Python package index (PyPI). Python Software Foundation; 2003, https://pypi.org.
[3] conda-forge community. The conda-forge Project: Community-based Software Distribution Built on the conda Package Format and Ecosystem. 2015, http://dx.doi.org/10.5281/zenodo.4774217, Zenodo.
[4] Coghlan N, Stufft D. Version identification and dependency specification. PEP 440, Python Software Foundation; 2013.
[5] Pimentel JF, Murta L, Braganholo V, Freire J. A large-scale study about quality and reproducibility of Jupyter notebooks. In: 2019 IEEE/ACM 16th international conference on mining software repositories. IEEE; 2019, p. 507–17. http://dx.doi.org/10.1109/MSR.2019.00077.
[6] Cannon B, Smith N, Stufft D. Specifying minimum build system requirements for Python projects. PEP 518, Python Software Foundation; 2016.
[7] Bicking I, Gábor B, Python Packaging Authority. virtualenv: Virtual Python environment builder. 2007, https://github.com/pypa/virtualenv.
[8] Eustace S. Poetry: Python packaging and dependency management made easy. 2019, https://github.com/python-poetry/poetry.
[9] Anaconda, Inc. conda. 2012, https://docs.conda.io.
[10] Merkel D. Docker: Lightweight Linux containers for consistent development and deployment. Linux J 2014;239(2):2.
[11] Kluyver T, Ragan-Kelley B, Pérez F, Granger B, Bussonnier M, Frederic J, et al. Jupyter notebooks – a publishing format for reproducible computational workflows. In: Loizides F, Scmidt B, editors. Positioning and power in academic publishing: players, agents and agendas. Netherlands: IOS Press; 2016, p. 87–90. http://dx.doi.org/10.3233/978-1-61499-649-1-87.
[12] Goldberg RP. Survey of virtual machine research. Computer 1974;7(6):34–45.
[13] Altintas Y, Brecher C, Weck M, Witt S. Virtual machine tool. CIRP Ann 2005;54(2):115–38. http://dx.doi.org/10.1016/S0007-8506(07)60022-5.
[14] Rosenblum M. VMware's Virtual Platform: A virtual machine monitor for commodity PCs. In: IEEE hot chips symposium. IEEE; 1999, p. 185–96.
[15] Kurtzer GM, Sochat V, Bauer MW. Singularity: Scientific containers for mobility of compute. PLoS One 2017;12(5):e0177459. http://dx.doi.org/10.1371/journal.pone.0177459.
[16] Martin G. A clash of kings. In: A Song of ice and fire, Voyager Books; 1998.
[17] Pérez F, Granger BE. IPython: A system for interactive scientific computing. Comput Sci Eng 2007;9(3):21–9. http://dx.doi.org/10.1109/MCSE.2007.53.
[18] van Rossum G, Lehtosalo J, Langa Ł. Type Hints. PEP 484, Python Software Foundation; 2014.
[19] Torvalds L, Hamano J. Git: Fast version control system. 2005, https://git.kernel.org/pub/scm/git/git.git.
[20] Harris CR, Millman KJ, van der Walt SJ, Gommers R, Virtanen P, Cournapeau D, et al. Array programming with NumPy. Nature 2020;585(7825):357–62. http://dx.doi.org/10.1038/s41586-020-2649-2.
[21] Smith EV. Data classes. PEP 557, Python Software Foundation; 2017.
[22] Heimes C, Cannon B. Removing dead batteries from the standard library. PEP 594, Python Software Foundation; 2019.
[23] McKinney W. Data structures for statistical computing in Python. In: van der Walt S, Millman J, editors. Proceedings of the 9th Python in science conference. 2010, p. 56–61. http://dx.doi.org/10.25080/Majora-92bf1922-00a.
[24] Pedregosa F, Varoquaux G, Gramfort A, Michel V, Thirion B, Grisel O, et al. Scikit-learn: Machine learning in Python. J Mach Learn Res 2011;12:2825–30.
[25] Varoquaux G. Joblib: Computing with Python functions. 2010, https://github.com/joblib/joblib.
[26] Abadi M, Agarwal A, Barham P, Brevdo E, Chen Z, Citro C, et al. TensorFlow: Large-Scale Machine Learning on Heterogeneous Systems. 2015, URL https://www.tensorflow.org/.
[27] McInnes L, Healy J, Saul N, Großberger L. UMAP: Uniform manifold approximation and projection. J Open Source Softw. 2018;3(29):861. http://dx.doi.org/10.21105/joss.00861.
[28] Hunter J. Matplotlib: A 2D graphics environment. Comput Sci Eng 2007;9(3):90–5. http://dx.doi.org/10.1109/MCSE.2007.55.
[29] Waskom ML. seaborn: Statistical data visualization. J Open Source Softw 2021;6(60):3021. http://dx.doi.org/10.21105/joss.03021.
[30] Heusser AC, Fitzpatrick PC, Field CE, Ziman K, Manning JR. Quail: A Python toolbox for analyzing and plotting free recall data. J Open Source Softw 2017;2(18). http://dx.doi.org/10.21105/joss.00424.
[31] Frederic J, Grout J, Jupyter Widgets Contributors. ipywidgets: Interactive Widgets for the Jupyter Notebook. 2015, https://github.com/jupyter-widgets/ipywidgets.
[32] da Costa-Luis C, Larroque SK, Altendorf K, Mary H, richardsheridan, Korobov M, et al. tqdm: A fast, extensible progress bar for Python and CLI. 2022, http://dx.doi.org/10.5281/zenodo.595120, https://github.com/tqdm/tqdm.
[33] Blei DM, Ng AY, Jordan MI. Latent dirichlet allocation. J Mach Learn Res 2003;3:993–1022.
[34] scikit-learn developers. scikit-learn User Guide: 9. Model persistence. 2022, https://scikit-learn.org/1.1/model_persistence.html.
[35] Manning JR, Whitaker EC, Fitzpatrick PC, Lee MR, Frantz AM, Bollinger BJ, et al. Feature and order manipulations in a free recall task affect memory for current and future lists. 2023, http://dx.doi.org/10.31234/osf.io/erzfp, PsyArXiv.
[36] Owen LLW, Manning JR. High-level cognition is supported by information-rich but compressible brain activity patterns. 2023, http://dx.doi.org/10.1101/2023.03.17.533152, bioRxiv.
[37] Ziman K, Lee MR, Martinez AR, Adner ED, Manning JR. Category-based and location-based volitional covert attention affect memory at different timescales. 2023, http://dx.doi.org/10.31234/osf.io/2ps6e, PsyArXiv.
[38] Manning JR. Storytelling with data. 2021, http://dx.doi.org/10.5281/zenodo.5182775, Zenodo, https://github.com/ContextLab/storytelling-with-data.
[39] Manning JR. ContextLab/experimental-psychology: v1.0 (Spring, 2022). 2022, http://dx.doi.org/10.5281/zenodo.6596762, Zenodo, https://github.com/ContextLab/experimental-psychology/tree/v1.0.
[40] MIND Team. Methods in Neuroscience at Dartmouth (MIND) computational summer school. 2023, https://mindsummerschool.org.
[41] Manning JR. abstract2paper. 2021, http://dx.doi.org/10.5281/zenodo.7261831, https://github.com/ContextLab/abstract2paper.
[42] Gao L, Biderman S, Black S, Golding L, Hoppe T, Foster C, et al. The pile: An 800GB dataset of diverse text for language modeling. 2020, http://dx.doi.org/10.48550/arXiv.2101.00027, arXiv preprint.
[43] Black S, Gao L, Wang P, Leahy C, Biderman S. GPT-Neo: Large Scale autoregressive language modeling with mesh-tensorflow. 2021, http://dx.doi.org/10.5281/zenodo.5297715, http://github.com/eleutherai/gpt-neo.
[44] van Veigen T, Akrami A, Bonnen K, DeWitt E, Hyafil A, Ledmyr H, et al. Neuromatch academy: Teaching computational neuroscience with global accessibility. Trends in Cognitive Sciences 2021;25(7):535–8. http://dx.doi.org/10.1016/j.tics.2021.03.018.
[45] Manning JR, Manjunatha H, Körding KP. Chatify: A jupyter extension for adding LLM-driven chatbots to interactive notebooks. 2023, http://dx.doi.org/10.5281/zenodo.8152315, https://github.com/ContextLab/chatify.
[46] Heusser AC, Ziman K, Owen LLW, Manning JR. HyperTools: A Python toolbox for gaining geometric insights into high-dimensional data. J Mach Learn Res 2018;18(152):1–6.
[47] Shannon M. Syntactic macros. PEP 638, Python Software Foundation; 2020.