Package 'basilisk'

February 1, 2026

Version 1.23.0

Date 2025-05-19

Title Freezing Python Dependencies Inside Bioconductor Packages

Depends reticulate

Imports utils, methods, parallel, dir.expiry

Suggests knitr, rmarkdown, BiocStyle, testthat, callr

biocViews Infrastructure

Description Installs a self-
contained conda instance that is managed by the R/Bioconductor installation machinery.
This aims to provide a consistent Python environment that can be used reliably by Bioconductor packages.
Functions are also provided to enable smooth interoperability of multiple Python environments in a single R session.

License GPL-3

RoxygenNote 7.3.2

StagedInstall false

VignetteBuilder knitr

BugReports https://github.com/LTLA/basilisk/issues

Encoding UTF-8

git_url https://git.bioconductor.org/packages/basilisk

git_branch devel

git_last_commit b653397

git_last_commit_date 2025-10-29

Repository Bioconductor 3.23

Date/Publication 2026-02-01

Author Aaron Lun [aut, cre, cph],
Vince Carey [ctb]

Maintainer Aaron Lun <infinite.monkeys.with.keyboards@gmail.com>

Contents

`BasiliskEnvironment-class` *The BasiliskEnvironment class*

Description

The BasiliskEnvironment class provides a simple structure containing all of the information to construct a **basilisk** environment. It is used by `basiliskStart` to perform lazy installation.

Constructor

`BasiliskEnvironment(envname, pkgname, packages)` will return a BasiliskEnvironment object, given:

- `envname`, string containing the name of the environment. Environment names starting with an underscore are reserved for internal use.
- `pkgname`, string containing the name of the package that owns the environment.
- `packages`, character vector containing the names of the required Python packages from PyPI. see `setupBasiliskEnv` for requirements.
- `channels`, deprecated and ignored.
- `pip`, appended to `packages`. Provided for back-compatibility only.
- `paths`, character vector containing relative paths to Python packages to be installed via `pip`. These paths are interpreted relative to the system directory of `pkgname`, i.e., they are appended to the output of `system.file` to obtain absolute paths for `setupBasiliskEnv`. Thus, any Python package vendored into the R package should be stored in the `inst` directory of the latter's source.

Author(s)

Aaron lun

Examples

```
BasiliskEnvironment("my_env1", "AaronPackage",
    packages=c("scikit-learn=1.6.1", "pandas=2.2.3"))
```

---

basiliskStart — *Start and stop* **basilisk***-related processes*

Description

Creates a **basilisk** process in which Python operations (via **reticulate**) can be safely performed with the correct versions of Python packages.

Usage

```
basiliskStart(
  env,
  full.activation = NA,
  fork = getBasiliskFork(),
  shared = getBasiliskShared(),
  testload = NULL
)

basiliskStop(proc)

basiliskRun(
  proc = NULL,
  fun,
  ...,
  env,
  full.activation = NA,
  persist = FALSE,
  fork = getBasiliskFork(),
  shared = getBasiliskShared(),
  testload = NULL
)
```

Arguments

`env`	A BasiliskEnvironment object specifying the **basilisk** environment to use. Alternatively, a string specifying the path to an environment, though this should only be used for testing purposes.
`full.activation`	Deprecated and ignored.

`fork`	Logical scalar indicating whether forking should be performed on non-Windows systems, see `getBasiliskFork`. If `FALSE`, a new worker process is created using communication over sockets.
`shared`	Logical scalar indicating whether `basiliskStart` is allowed to load a shared Python instance into the current R process, see `getBasiliskShared`.
`testload`	Deprecated and ignored.
`proc`	A process object generated by `basiliskStart`.
`fun`	A function to be executed in the **basilisk** process, see “Constraints on user-defined functions”.
`...`	Further arguments to be passed to `fun`.
`persist`	Logical scalar indicating whether to pass a persistent store to `fun`. If `TRUE`, `fun` should accept a `store` argument.

Details

These functions ensure that any Python operations in `fun` will use the environment specified by `envname`. This avoids version conflicts in the presence of other Python instances or environments loaded by other packages or by the user. Thus, **basilisk** clients are not affected by (and if `shared=FALSE`, do not affect) the activity of other R packages.

It is good practice to call `basiliskStop` once computation is finished to terminate the process. Any Python-related operations between `basiliskStart` and `basiliskStop` should only occur via `basiliskRun`. Calling **reticulate** functions directly will have unpredictable consequences, Similarly, it would be unwise to interact with `proc` via any function other than the ones listed here.

If `proc=NULL` in `basiliskRun`, a process will be created and closed automatically. This may be convenient in functions where persistence is not required. Note that doing so requires specification of `pkgname` and `envname`.

Value

`basiliskStart` returns a process object, the exact nature of which depends on `fork` and `shared`. This object should only be used in `basiliskRun` and `basiliskStop`.

`basiliskRun` returns the output of `fun(...)`, possibly executed inside the separate process.

`basiliskStop` stops the process in `proc`.

Choice of process type

- If `shared=TRUE` and no Python version has already been loaded, `basiliskStart` will load Python directly into the R session from the specified environment. Similarly, if the existing environment is the same as the requested environment, `basiliskStart` will use that directly. This mode is most efficient as it avoids creating any new processes, but the use of a shared Python configuration may prevent non-**basilisk** packages from working correctly in the same session.
- If `fork=TRUE`, no Python version has already been loaded and we are not on Windows, `basiliskStart` will create a new process by forking. In the forked process, `basiliskStart` will load the specified environment for operations in Python. This is less efficient as it needs to create a new process but it avoids forcing a Python configuration on other packages in the same R session.

- Otherwise, `basiliskStart` will create a parallel socket process containing a separate R session. In the new process, `basiliskStart` will load the specified environment for Python operations. This is the least efficient as it needs to transfer data over sockets but is guaranteed to work.

Developers can control these choices directly by explicitly specifying `shared` and `fork`, while users can control them indirectly with `setBasiliskFork` and related functions.

Constraints on user-defined functions

In `basiliskRun`, there is no guarantee that `fun` has access to `basiliskRun`'s calling environment. This has several consequences for code in the body of `fun`:

- Variables used inside `fun` should be explicitly passed as an argument to `fun`. Developers should not rely on closures to capture variables in the calling environment of `basiliskRun`.
- Developers should *not* attempt to pass complex objects to memory in or out of `fun`. This mostly refers to objects that contain custom pointers to memory, e.g., file handles, pointers to **reticulate** objects. Both the arguments and return values of `fun` should be pure R objects.
- Functions or variables from non-base R packages should be prefixed with the package name via `::`, or those packages should be reloaded inside `fun`.

Developers can test that their function behaves correctly in `basiliskRun` by setting `setBasiliskShared` and `setBasiliskFork` to `FALSE`. This forces the execution of `fun` in a new process; any incorrect assumption of shared environments will cause errors.

Persisting objects across calls

Objects created inside `fun` can be persisted across calls to `basiliskRun` by setting `persist=TRUE`. This will instruct `basiliskRun` to pass a `store` argument to `fun` that can be used to store arbitrary objects. Those same objects can be retrieved from `store` in later calls to `basiliskRun` using the same `proc`. Any object can be stored in `.basilisk.store` but will remain strictly internal to `proc`.

This capability is primarily useful when a Python workflow is split across multiple `basiliskRun` calls. Each subsequent call can pick up from temporary intermediate objects generated by the previous call. In this manner, **basilisk** enables modular function design where developers can easily mix and match different `basiliskRun` invocations. See Examples for a working demonstration.

Use of lazy installation

If the specified **basilisk** environment is not present and `env` is a BasiliskEnvironment object, the environment will be created upon first use of `basiliskStart`. These environments are created in an external user-writable directory defined by `getExternalDir`. The location of this directory can be changed by setting the `BASILISK_EXTERNAL_DIR` environment variable to the desired path. This may occasionally be necessary in rare cases, e.g., if the file path to the default location is too long for Windows.

Advanced users may consider setting the environment variable `BASILISK_USE_SYSTEM_DIR` to 1 when installing **basilisk** client packages from source. This will place each package's Python environments in the R system directory, which simplifies permission management and avoids duplication in enterprise settings.

Author(s)

Aaron Lun

See Also

setupBasiliskEnv, to set up the Python environments.

getBasiliskFork and getBasiliskShared, to control various global options.

Examples

```
tmploc <- file.path(tempdir(), "my_package_A")
if (!file.exists(tmploc)) {
    setupBasiliskEnv(tmploc, c('pandas=2.2.3'))
}

# Pulling out the pandas version, as a demonstration:
cl <- basiliskStart(tmploc)
basiliskRun(proc=cl, function() {
    X <- reticulate::import("pandas"); X$`__version__`
})
basiliskStop(cl)

# This happily co-exists with our other environment:
tmploc2 <- file.path(tempdir(), "my_package_B")
if (!file.exists(tmploc2)) {
    setupBasiliskEnv(tmploc2, c('pandas=2.2.2'))
}

cl2 <- basiliskStart(tmploc2)
basiliskRun(proc=cl2, function() {
    X <- reticulate::import("pandas"); X$`__version__`
})
basiliskStop(cl2)

# Persistence of variables is possible within a Start/Stop pair.
cl <- basiliskStart(tmploc)
basiliskRun(proc=cl, function(store) {
    store$snake.in.my.shoes <- 1
    invisible(NULL)
}, persist=TRUE)
basiliskRun(proc=cl, function(store) {
    return(store$snake.in.my.shoes)
}, persist=TRUE)
basiliskStop(cl)
```

clearExternalDir *Clear the external installation directory*

Description

Clear the external installation directory by removing all or obsolete virtual environments. This can be used to free up some space if the expiry mechanism is not fast enough at deleting unused environments.

Usage

```
clearExternalDir(
  path = getExternalDir(),
  package = NULL,
  obsolete.only = FALSE
)
```

Arguments

`path`	String containing a path to the external directory containing the virtual environments.
`package`	String containing the name of a client R package. If provided, all environments will be removed for this package.
`obsolete.only`	Logical scalar indicating whether to only remove environments for obsolete package versions.

Value

If `package=NULL` and `obsolete.only=FALSE`, all of the virtual environments in the external directory are destroyed. If `obsolete.only=TRUE`, the environments associated with **basilisk** versions older than the current version are destroyed.

If `package` is supplied and `obsolete.only=FALSE`, all virtual environments for the specified client package are destroyed. If `obsolete.only=FALSE`, only the environments for older versions of the client package are destroyed.

Author(s)

Aaron Lun

See Also

`getExternalDir`, which determines the location of the external directory.

Examples

```
# We can't actually run clearExternalDir() here, as it
# relies on basilisk already being installed.
print("dummy test to pass BiocCheck")
```

`configureBasiliskEnv` *Configure client environments*

Description

Configure the **basilisk** environments in the `configure` file of client packages.

Usage

```
configureBasiliskEnv(src = "R/basilisk.R")
```

Arguments

`src`	String containing path to a R source file that defines one or more BasiliskEnvironment objects.

Details

This function is designed to be called in the `configure` file of client packages, triggering the construction of **basilisk** environments during package installation. It will only run if the `BASILISK_USE_SYSTEM_DIR` environment variable is set to `"1"`.

We take a source file as input to avoid duplicated definitions of the BasiliskEnvironments. These objects are used in `basiliskStart` in the body of the package, so they naturally belong in `R/`; we then ask `configure` to pull out that file (named `"basilisk.R"` by convention) to create these objects during installation.

The source file in `src` should be executable on its own, i.e., you can `source` it without loading any other packages (beside **basilisk**, obviously). Non-BasiliskEnvironment objects can be created but are simply ignored in this function.

Value

One or more **basilisk** environments are created corresponding to the BasiliskEnvironment objects in `src`. A `NULL` is invisibly returned.

Author(s)

Aaron Lun

See Also

`setupBasiliskEnv`, which does the heavy lifting of setting up the environments.

Examples

```
## Not run:
configureBasiliskEnv()

## End(Not run)
```

`createLocalBasiliskEnv` *Manually create a local virtual environment manually*

Description

Manually create a local virtual environment with versioning and thread safety. This is intended for use in analysis workflows rather than package development.

Usage

```
createLocalBasiliskEnv(dir, ...)
```

Arguments

`dir`	String containing the path to a directory in which the local environment is to be stored.
`...`	Further arguments to pass to `setupBasiliskEnv`.

Details

This function is intended for end users who wish to use the **basilisk** machinery for coordinating one or more Python environments in their analysis workflows. It can be inserted into, e.g., Rmarkdown reports to automatically provision and cache an environment on first compilation, which will be automatically re-used in later compilations. Some care is taken to ensure that the cached environment is refreshed when **basilisk** is updated, and that concurrent access to the environment is done safely.

Value

String containing a path to the newly created environment, or to an existing environment if one was previously created. This can be used in `basiliskRun`.

Author(s)

Aaron Lun

Examples

```
tmploc <- file.path(tempdir(), "my_package_C")
tmp <- createLocalBasiliskEnv(tmploc, packages="pandas=2.2.3")
basiliskRun(env=tmp, fun=function() {
    X <- reticulate::import("pandas"); X$`__version__`
})
```

`destroyOldVersions` *Destroy old versions?*

Description

Should we destroy old environments for **basilisk** client packages?

Usage

```
destroyOldVersions()
```

Details

The default value is `TRUE`, in order to save some disk space. This can be changed by setting `BASILISK_NO_DESTROY` environment variable to `"1"`.

Value

Logical scalar providing an answer to the above.

Author(s)

Aaron Lun

See Also

`obtainEnvironmentPath`, which removes old environments as a side-effect.

`getBasiliskFork` *Options for* **basilisk**

Description

Options controlling run-time behavior of **basilisk**. Unlike the various environment variables, these options can be turned on or off by users without requiring reinstallation of the **basilisk** ecosystem.

Usage

```
getBasiliskFork()

setBasiliskFork(value)

getBasiliskShared()

setBasiliskShared(value)

setBasiliskCheckVersions(value)

getBasiliskCheckVersions()
```

Arguments

`value`	Logical scalar: • For `setBasiliskFork`, whether forking should be used when available. • For `setBasiliskShared`, whether the shared Python instance can be set in the R session. • For `setBasiliskCheckVersions`, whether to check for properly versioned package strings in `setupBasiliskEnv`.

Value

All functions return a logical scalar indicating whether the specified option is enabled.

Controlling process creation

By default, `basiliskStart` will attempt to load a shared Python instance into the R session. This avoids the overhead of setting up a new process but will potentially break any **reticulate**-dependent code outside of **basilisk**. To guarantee that non-**basilisk** code can continue to execute, users can set `setBasiliskShared(FALSE)`. This will load the Python instance into a self-contained **basilisk** process.

If a new process must be generated by `basiliskStart`, forking is used by default. This is generally more efficient than socket communication when it is available (i.e., not on Windows), but can be less efficient if any garbage collection occurs inside the new process. In such cases, users or developers may wish to turn off forking with `setBasiliskFork(FALSE)`, e.g., in functions where many R-based memory allocations are performed inside `basiliskRun`.

If many **basilisk**-dependent packages are to be used together on Unix systems, setting `setBasiliskShared(FALSE)` may be beneficial. This allows each package to fork to create a new process as no Python has been loaded in the parent R process (see ?`basiliskStart`). In contrast, if any package loads Python sharedly, the others are forced to use parallel socket processes. This results in a tragedy of the commons where the efficiency of all other packages is reduced.

Developers may wish to set `setBasiliskShared(FALSE)` and `setBasiliskFork(FALSE)` during unit testing, to ensure that their functions do not make incorrect assumptions about the calling environment used in `basiliskRun`.

Disabling package version checks

By default, `setupBasiliskEnv` requires versions for all requested Python packages. However, in some cases, the exact version of the packages may not be known beforehand. Developers can set `setBasiliskCheckVersions(FALSE)` to disable all version checks, instead allowing `pip` to choose appropriate versions for the initial installation. The resulting environment can then be queried using `listPackages` to obtain the explicit versions of all Python packages.

Needless to say, this option should only be used during the initial phases of developing a **basilisk** client. Once a suitable environment is created, Python package versions should be pinned in `setupBasiliskEnv`. This ensures that all users are creating the intended environment for greater reproducibility (and easier debugging).

Author(s)

Aaron Lun

See Also

`basiliskStart`, where these options are used.

Examples

```
getBasiliskFork()
getBasiliskShared()
```

`getExternalDir` *External directory for virtual environments*

Description

Define an external location for installing the virtual environments managed by **basilisk**.

Usage

```
getExternalDir()
```

Details

The default path contains the version number so that installation of a new version of **basilisk** will trigger the creation of new virtual environments. This ensures that any changes to **basilisk** functions will be respected by all client packages in the same R installation.

If the `BASILISK_EXTERNAL_DIR` environment variable is set to some absolute path, this will be used instead as the installation directory. Setting this variable is occasionally necessary if the default path returned by `R_user_dir` has spaces; or on Windows, if the 260 character limit is exceeded after combining the default path with deeply nested environment paths.

We assume that the user has read-write access to the external directory. Write access is necessary to generate new environments and to handle locking in `lockExternalDir`.

Value

String containing a path to an appropriate external folder. The last component of the path will always be the **basilisk** version number.

Author(s)

Aaron Lun

Examples

```
# We can't actually run getExternalDir() here, as it
# either relies on basilisk already being installed.
print("dummy test to pass BiocCheck")
```

`getPythonBinary` *Get Python binary paths*

Description

Get Python binary paths

Usage

```
getPythonBinary(envpath)
```

Arguments

`envpath`	String containing the path to a virtual environment.

Details

This code is largely copied from **reticulate**, and is only present here as they do not export these utilities for general consumption.

Value

String containing the path to the Python executable inside `envpath`.

Author(s)

Aaron Lun

Examples

```
getPythonBinary("foo/bar")
```

`getSystemDir` *Get the system installation directory*

Description

Get the system installation directory for a package. This is not entirely trivial as it may be called before the package is installed.

Usage

```
getSystemDir(pkgname, installed)
```

Arguments

`pkgname`	String containing the package name.
`installed`	Logical scalar specifying whether the package is likely to be installed yet.

Value

String containing the path to the (likely, if `installed=FALSE`) installation directory for `pkgname`.

Author(s)

Aaron Lun

Examples

```
getSystemDir("basilisk", installed=FALSE)
```

isWindows — *Find the operating system or architecture.*

Description

Indicate whether we are on Windows or MacOSX. For MacOSX and Linux, we can also determine if we are on an x86-64 or Arm-based architecture.

Usage

```
isWindows()

isMacOSX()

isMacOSXArm()

isLinux()

isLinuxAarch64()
```

Value

Logical scalar indicating whether we are on the specified OS and/or architecture.

Author(s)

Aaron Lun

Examples

```
isWindows()
isMacOSX()
isLinux()
```

listPackages — *List packages*

Description

List the set of Python packages (and their version numbers) that are installed in a virtual environment.

Usage

```
listPackages(env)

listPythonVersion(env)
```

Arguments

`env`	A BasiliskEnvironment object specifying the **basilisk** environment to use. Alternatively, a string specifying the path to an environment, though this should only be used for testing purposes.

Value

For `listPackages`, a data.frame containing the `full`, a versioned package string, and `package`, the package name.

For `listPythonVersion`, a string containing the default version of Python.

Author(s)

Aaron Lun

Examples

```
tmploc <- file.path(tempdir(), "my_package_A")
if (!file.exists(tmploc)) {
    setupBasiliskEnv(tmploc, c('pandas=1.4.3'))
}
listPackages(tmploc)
listPythonVersion(tmploc)
```

`lockExternalDir` *Lock external directory*

Description

Lock the external directory so that multiple processes cannot try to create environments at the same time.

Usage

```
lockExternalDir(path = getExternalDir(), ...)

unlockExternalDir(lock.info, ...)
```

Arguments

`path`	String containing the path to the external directory.
`...`	For `lockExternalDir`, further arguments to pass to `lockDirectory` such as `exclusive`. For `unlockExternalDir`, further arguments to pass to `unlockDirectory` such as `clear`.
`lock.info`	A lock object generated by `lockDirectory`.

Details

This will apply a lock to the (possibly user-specified) external environment directory, so that a user trying to run parallel **basilisk** processes will not have race conditions during lazy environment creation. We use **dir.expiry** to manage the locking process for us, with the following strategy:

- If a system installation is being performed, we do not perform any locking. Rather, the R package manager will lock the entire R installation directory for us.
- If the external directory is not yet present, we establish an exclusive lock and create that directory.
- If an external installation directory is already present, we establish a shared lock. This will wait for any exclusive lock to expire (and thus any currently running installation to finish). No waiting is required if there are no existing exclusive locks.

Note that locking is only required during the creation of virtual environments, not during their actual use. Once an environment is created, we assume that it is read-only for all processes.

Value

`lockExternalDir` will return a lock object from `lockDirectory`.

`unlockExternalDir` will unlock the file and return `NULL` invisibly.

Author(s)

Aaron Lun

Examples

```
loc <- lockExternalDir()
unlockExternalDir(loc)
```

obtainEnvironmentPath *Obtain the environment path*

Description

Obtain a path to a virtual environment, lazily creating it if it does not already exist.

Usage

```
obtainEnvironmentPath(env)
```

Arguments

env	A BasiliskEnvironment object specifying the environment. Alternatively a string containing a path to an existing environment.

Value

String containing the path to an instantiated virtual environment.

For a BasiliskEnvironment `env`, the function will also lazily create the environment, if `useSystemDir()` returns `FALSE` and the environment does not already exist.

Author(s)

Aaron Lun

Examples

```
tmploc <- file.path(tempdir(), "my_package_A")
if (!file.exists(tmploc)) {
    setupBasiliskEnv(tmploc, c('pandas=1.4.3'))
}
obtainEnvironmentPath(tmploc)

env <- BasiliskEnvironment("test_env", "basilisk",
    packages=c("scikit-learn=1.1.1", "pandas=1.43.1"))
## Not run: obtainEnvironmentPath(env)
```

`PyPiLink` *Link to PyPi*

Description

Helper function to create a Markdown link to the PyPi landing page for a Python package. Intended primarily for use inside vignettes.

Usage

```
PyPiLink(package)
```

Arguments

`package`	String containing the name of the Python package.

Value

String containing a Markdown link to the package's landing page.

Author(s)

Aaron Lun

Examples

```
PyPiLink("pandas")
PyPiLink("scikit-learn")
```

`setupBasiliskEnv` *Set up* **basilisk***-managed environments*

Description

Set up a virtual environment for isolated execution of Python code with appropriate versions of all Python packages.

Usage

```
setupBasiliskEnv(envpath, packages, channels = NULL, pip = NULL, paths = NULL)
```

Arguments

`envpath`	String containing the path to the environment to use.
`packages`	Character vector containing the names of PyPI packages to install into the environment. Version numbers must be included.
`channels`	Deprecated and ignored.
`pip`	Same as `packages`.
`paths`	Character vector containing absolute paths to Python package directories, to be installed by `pip`.

Details

Developers of **basilisk** client packages should never need to call this function directly. For typical usage, `setupBasiliskEnv` is automatically called by `basiliskStart` to perform lazy installation. Developers should also create `configure(.win)` files to call `configureBasiliskEnv`, which will call `setupBasiliskEnv` during R package installation when `BASILISK_USE_SYSTEM_DIR=1`.

Pinned version numbers must be present for all desired `packages`. This improves predictability and simplifies debugging across different systems. Any `=` version specifications will be automatically converted to `==`.

Additional Python packages can be installed from local directories via the `paths` argument. This is useful for **basilisk** clients vendoring Python packages that are not available in standard repositories. While `paths` expects absolute paths for general usage, this will be auto-generated in a package development context - see `BasiliskEnvironment` for details.

It is also good practice to explicitly list the versions of the *dependencies* of all desired packages. This protects against future changes in the behavior of your code if the `pip` dependency resolver decides to use a different version of a dependency. To identify appropriate versions of dependencies, we suggest:

1. Creating a fresh virtual environment with the desired packages, using `packages=` in `setupBasiliskEnv`.
2. Calling `listPackages` on the environment to identify any relevant dependencies and their versions.
3. Including those dependencies in the `packages=` argument for future use. (It is helpful to mark dependencies in some manner, e.g., with comments, to distinguish them from the actual desired packages.)

If versions for the desired packages are not known beforehand, developers may use `setBasiliskCheckVersions(FALSE)` before running `setupBasiliskEnv`. This instructs `setupBasiliskEnv` to create an environment with appropriate versions of all unpinned packages, which can then be read out via `listPackages` for insertion in the `packages=` argument as described above. We stress that this option should *not* be used in any release of the R package, it is a development-phase-only utility.

The virtual environment can be created with a specific Python version by specifying `python=` in `packages`, e.g., `"python=3.10"`. If no Python version is listed, the environment is created with the `defaultPythonVersion` version. It is advised to explicitly list the desired version of Python in `packages`, even if this is already version-compatible with the default. This protects against changes to the Python version in future **basilisk** versions. Each requested Python version will be installed using Pyenv with `install_python` if it is not already available.

Instead of installing a new Python instance, administrators or users can use their own Python by defining the `BASILISK_CUSTOM_PYTHON_*` environment variable. This can be of the form `BASILISK_CUSTOM_PYTHON_X`, for Python version `X`; or `BASILISK_CUSTOM_PYTHON_X_Y`, for Python version `X.Y`; or `BASILISK_CUSTOM_PYTHON_X_Y_Z`, for Python version `X.Y.Z`. In each case, the environment variable should be set to a path to a Python binary. When a particular Python version is requested, `setupBasiliskEnv` will check the corresponding environment variable.

Administrators can also set the `BASILISK_NO_PYENV` environment variable to 1 to disable installation via Pyenv altogether. Doing so instructs `setupBasiliskEnv` to throw an error if a custom Python is not available via the `BASILISK_CUSTOM_PYTHON_*` variables. This can occasionally be useful to prevent unexpected installations of new Python instances.

Value

A virtual environment is created at `envpath` containing the specified `packages`. A `NULL` is invisibly returned.

See Also

`listPackages`, to list the packages in the virtual environment.

Examples

```
tmploc <- file.path(tempdir(), "my_package_A")
if (!file.exists(tmploc)) {
    setupBasiliskEnv(tmploc, c('pandas=2.2.3'))
}
```

`useBasiliskEnv` *Use* **basilisk** *environments*

Description

Use **basilisk** environments for isolated execution of Python code with appropriate versions of all Python packages.

Usage

```
useBasiliskEnv(envpath, full.activation = NA)
```

Arguments

`envpath`	String containing the path to the **basilisk** environment to use.
`full.activation`	Deprecated and ignored.

Details

useBasiliskEnv will load the Python binary at envpath (or specifically, its shared library) into the current R session via **reticulate**. Users can then use, e.g., import to access functionality in the Python packages installed in the virtual environment.

To ensure that the correct packages are used, useBasiliskEnv will unset environment variables like PYTHONPATH and PYTHONHOME. These can be restored by running the function returned by useBasiliskEnv.

It is unlikely that developers should ever need to call useBasiliskEnv directly. Rather, this interaction should be automatically handled by basiliskStart.

Value

The function will attempt to load the specified **basilisk** environment into the R session, possibly with the modification of some environment variables (see Details).

A function is invisibly returned that accepts no arguments and resets all environment variables to their original values prior to the useBasiliskEnv call.

Author(s)

Aaron Lun

See Also

basiliskStart, for how these **basilisk** environments should be used.

Examples

```
tmploc <- file.path(tempdir(), "my_package_A")
if (!file.exists(tmploc)) {
    setupBasiliskEnv(tmploc, c('pandas==2.2.3'))
}

# This may or may not work, depending on whether a Python instance
# has already been loaded into this R session.
try(useBasiliskEnv(tmploc))
```

---

useSystemDir *Use the R system directory?*

Description

Should we use the R system directory for installing Python environments in **basilisk** clients?

Usage

```
useSystemDir()
```

Details

The default value is `FALSE` to avoid problems with position-dependent code in packaged binaries. This can be changed by setting `BASILISK_USE_SYSTEM_DIR` environment variable to `"1"`.

Value

Logical scalar providing an answer to the above.

Author(s)

Aaron Lun

See Also

`link{configureBasiliskEnv}`, which is run during package installation.

Index