Package: basilisk (via r-universe)

June 30, 2024

Version 1.17.0

Date 2024-02-17

Title Freezing Python Dependencies Inside Bioconductor Packages

Depends reticulate

Imports utils, methods, parallel, dir.expiry, basilisk.utils (>= 1.15.1)

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 7.3.1

StagedInstall false

VignetteBuilder knitr

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

Encoding UTF-8

Repository https://bioc.r-universe.dev

RemoteUrl https://github.com/bioc/basilisk

RemoteRef HEAD

RemoteSha 1539dedd52953489277fdb3c1f57a3b9b6573121

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 conda, see `setupBasiliskEnv` for requirements.
- `channels`, character vector specifying the Conda channels to search.
- `pip`, character vector containing names of additional Python packages from PyPi, see `setupBasiliskEnv` for requirements.
- `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`.

Author(s)

Aaron lun

Examples

```
BasiliskEnvironment("my_env1", "AaronPackage",
    packages=c("scikit-learn=1.1.1", "pandas=1.43.1"))
```

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. Alternatively, NULL to indicate that the base Conda installation should be used as the environment.
full.activation	Logical scalar, see activateEnvironment for more details.
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	Character vector specifying the Python packages to load into the process during set-up. This is used to check that packages can be correctly loaded, switching to a fallback on GLIBCXX dynamic linking failures.
proc	A process object generated by basiliskStart.
fun	A function to be executed in the **basilisk** process. This should return a "pure R" object, see details.
...	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.

If the base Conda installation provided with **basilisk** satisfies the requirements of the client package, developers can set env=NULL in this function to use that base installation rather than constructing a separate environment.

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.

Testing package loads

If `testload` is provided, `basiliskStart` will attempt to load those Python packages into the newly created process. This is used to detect loading failures due to differences in the versions of the shared libraries. Most typically, a conda-supplied Python package (often **scipy** submodules) will have been compiled against a certain version of `libstdc++` but R is compiled against an older version. R's version takes precedence when **reticulate** attempts to load the Python package, causing cryptic "GLIBCXX version not found" errors.

By checking the specified `testload`, `basiliskStart` can check for loading failures in potentially problematic packages. Upon any failure, `basiliskStart` will fall back to a separate socket process running a conda-supplied R installation. The idea is that, if both Python and R are sourced from conda, they will be using the same version of `libstdc++` and other libraries. This avoids loading errors and/or segmentation faults due to version mismatches.

Use of this "last resort fallback" overrides any choice of process type from `fork` and `shared`. If no failures are encountered, a process will be created using the current R installation.

Note that the fallback R installation is very minimalistic; only **reticulate** is guaranteed to be available. This places some limitations on the code that can be executed inside `fun` for **basilisk** environments that might trigger use of the fallback.

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`. However, if `fun` loads Python packages that might trigger the last resort fallback, no functions or variables should be used from non-base R packages.

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. If `fun` involves fallback-inducing Python packages, developers can further set `setBasiliskForceFallback` before running `basiliskRun`. This tests that `fun` works with the minimal conda-supplied R installation.

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`. If the base Conda installation is not present, it will also be installed upon first use of `basiliskStart`. We do not provide Conda with the **basilisk** package binaries to avoid portability problems with hard-coded paths (as well as potential licensing issues from redistribution).

By default, both the base conda installation and the environments will be placed in an external user-writable directory defined by **rappdirs** via `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 if the file path to the default location is too long for Windows, or if the default path has spaces that break the Miniconda/Anaconda installer.

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

Persistence of environment variables

When `shared=TRUE` and if no Python instance has already been loaded into the current R session, a side-effect of `basiliskStart` is that it will modify a number of environment variables. This is done to mimic activation of the Conda environment located at `env`. Importantly, old values for these variables will *not* be restored upon `basiliskStop`.

This behavior is intentional as (i) the correct use of the Conda-derived Python depends on activation and (ii) the loaded Python persists for the entire R session. It may not be safe to reset the environment variables and "deactivate" the environment while the Conda-derived Python instance is effectively still in use. (In practice, lack of activation is most problematic on Windows due to its dependence on correct `PATH` specification for dynamic linking.)

If persistence is not desirable, users should set `shared=FALSE` via `setBasiliskShared`. This will limit any modifications to the environment variables to a separate R process.

Author(s)

Aaron Lun

See Also

setupBasiliskEnv, to set up the conda environments.

activateEnvironment in the **basilisk.utils** package.

getBasiliskFork and getBasiliskShared, to control various global options.

Examples

```
if (.Platform$OS.type != "windows") {


 # Creating an environment (note, this is not necessary
 # when supplying a BasiliskEnvironment to basiliskStart):
 tmploc <- file.path(tempdir(), "my_package_A")
 if (!file.exists(tmploc)) {
     setupBasiliskEnv(tmploc, c('pandas=1.4.3'))
 }

 # Pulling out the pandas version, as a demonstration:
 cl <- basiliskStart(tmploc, testload="pandas")
 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=1.4.2'))
 }

 cl2 <- basiliskStart(tmploc2, testload="pandas")
 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)
}
```

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` *Create a local conda environment manually*

Description

Manually create a local conda 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

```
if (.Platform$OS.type != "windows") {
 tmploc <- file.path(tempdir(), "my_package_C")
 tmp <- createLocalBasiliskEnv(tmploc, packages="pandas==1.4.3")
 basiliskRun(env=tmp, fun=function() {
     X <- reticulate::import("pandas"); X$`__version__`
 }, testload="pandas")
}
```

---

findPersistentEnv *Find the persistent environment*

Description

Previously used to find a persistent environment inside a basiliskRun call, to allow variables to be passed across calls. This is deprecated in favor of setting `persist=TRUE` in basiliskRun, which will explicitly pass an environment in which to store persistent variables.

Usage

```
findPersistentEnv()
```

Value

A defunct error message is raised.

Author(s)

Aaron Lun

---

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)

getBasiliskForceFallback()

setBasiliskForceFallback(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 `setBasiliskForceFallback`, whether to force the use of the last resort fallback. • 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`. Similarly, setting `setBasiliskForceFallback(TRUE)` is useful for testing that `basiliskRun` works inside a minimalistic R installation.

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 conda 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 by conda, 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()
```

---

listPackages *List packages*

Description

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

Usage

```
listPackages(env = NULL)

listCorePackages()

listPythonVersion(env = 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. Alternatively, NULL to indicate that the base Conda installation should be used as the environment.

Details

This is provided for informational purposes only; developers should not expect the same core packages to be present across operating systems. ?installConda has some more comments on the version of the conda installer used for each operating system.

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

```
listPackages()
listPythonVersion()
```

---

obtainEnvironmentPath *Obtain the environment path*

Description

Obtain a path to a Conda environment, lazily installing it if necessary.

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 Conda 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

```
if (.Platform$OS.type != "windows") {


 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 conda environment for isolated execution of Python code with appropriate versions of all Python packages.

Usage

```
setupBasiliskEnv(
  envpath,
  packages,
  channels = "conda-forge",
  pip = NULL,
  paths = NULL
)
```

Arguments

`envpath`	String containing the path to the environment to use.
`packages`	Character vector containing the names of conda packages to install into the environment. Version numbers must be included.
`channels`	Character vector containing the names of additional conda channels to search. Defaults to the conda-forge repository.
`pip`	Character vector containing the names of additional packages to install from PyPi using `pip`. Version numbers must be included.
`paths`	Character vector containing absolute paths to Python package directories, to be installed by `pip`.

Details

Developers of 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 conda packages in `packages`. This improves predictability and simplifies debugging across different systems. Note that the version notation for conda packages uses a single `=`, while the notation for Python packages uses `==`; any instances of the latter will be coerced to the former automatically.

It is possible to use the pip argument to install additional packages from PyPi after all the conda packages are installed. All packages listed here are also expected to have pinned versions, this time using the == notation. However, some caution is required when mixing packages from conda and pip, see https://www.anaconda.com/using-pip-in-a-conda-environment for more details.

It is further possible to install Python packages from directories. In the package development context, this typically assumes that the Python directories are included in the inst subdirectory of the R package. basiliskStart will then convert the relative path to an absolute path before calling this function - 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 conda's solver decides to use a different version of a dependency. To identify appropriate versions of dependencies, we suggest:

1. Creating a fresh conda 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.)

The only reason that pinned dependencies are not mandatory is because some dependencies are OS-specific, requiring some manual pruning of the output of listPackages.

If versions for the desired conda packages are not known beforehand, developers may use setBasiliskCheckVersions(FALS before running setupBasiliskEnv. This instructs conda 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.

If no Python version is listed, the version in the base conda installation is used by default - check listPythonVersion for the version number. However, it is often prudent to explicitly list the desired version of Python in packages, even if this is already version-compatible with the default (e.g., "python=3.8"). This protects against changes to the Python version in the base installation, e.g., if administrators override the choice of conda installation with certain environment variables. Of course, it is possible to specify an entirely different version of Python in packages by supplying, e.g., "python=2.7.10".

Value

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

See Also

listPackages, to list the packages in the conda environment.

Examples

```
if (.Platform$OS.type != "windows") {
```

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

---

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	Logical scalar, see activateEnvironment for details.

Details

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

This function will modify a suite of environment variables as a side effect - see "Persistence of environment variables" in ?basiliskStart for the rationale.

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 NULL is invisibly returned.

Author(s)

Aaron Lun

See Also

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

Examples

```
if (.Platform$OS.type != "windows") {


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

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

 # This will definitely not work, as the available Python is already set.
 baseloc <- basilisk.utils::getCondaDir()
 status <- try(useBasiliskEnv(baseloc))

 # ... except on Windows, which somehow avoids tripping the error.
 stopifnot(is(status, "try-error") || basilisk.utils::isWindows())
}
```

Index