Package ‘basilisk’

October 16, 2020

Version 1.0.2

Date 2020-05-11

Title Freezing Python Versions Inside Bioconductor Packages

Imports reticulate, parallel, methods, basilisk.utils

Suggests knitr, rmarkdown, BiocStyle, testthat, callr

biocViews Infrastructure

Description Installs a self-contained Python instance that is managed by the R installation.
This aims to provide a consistent Python version that can be used reliably by Bioconductor packages.
Module versions are also controlled to guarantee consistent behavior on different user systems.

License GPL-3

RoxygenNote 7.1.0

StagedInstall false

VignetteBuilder knitr

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

git_branch RELEASE_3_11

git_last_commit 15b9066

git_last_commit_date 2020-05-11

Date/Publication 2020-10-16

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

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

R topics documented:

`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.
- `pip`, character vector containing names of additional Python packages from PyPi, see `setupBasiliskEnv` for requirements.

Author(s)

Aaron lun

Examples

```
BasiliskEnvironment("my_env1", "AaronPackage",
    packages=c("scikit-learn=0.22.0", "pandas=0.24.1"))
```

`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, fork = getBasiliskFork(), shared = getBasiliskShared())

basiliskStop(proc)

basiliskRun(
  proc = NULL,
  fun,
  ...,
  env,
  fork = getBasiliskFork(),
  shared = getBasiliskShared()
)
```

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.
`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`.
`proc`	A process object generated by `basiliskStart`.
`fun`	A function to be executed in the **basilisk** process.
`...`	Further arguments to be passed to `fun`.

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.

If necessary, objects created in `fun` can persist across calls to `basiliskRun`, e.g., for file handles. This requires the use of `assign` with `envir` set to `findPersistentEnv` to persist a variable, and a corresponding `get` to retrieve that object in later calls. See Examples for more details.

It is good practice to call `basiliskStop` once computation is finished. This will close the **basilisk** processes and restore certain environment variables to their original state (e.g., `"PYTHONPATH"`) so that other non-**basilisk** operations can operate properly.

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(...)` when executed inside the separate process.

`basiliskStop` stops the process in `proc`.

Choice of backend

- 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.
- Otherwise, 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.

If the base conda installation provided with **basilisk** satisfies the requirements of the client package, it is strongly recommended to set `env=NULL` rather than constructing a separate environment. This is obviously easier but it is also more efficient as it increases the chance of multiple **basilisk** clients being able to share a common Python instance within the same R session.

Constraints on user-defined functions

In `basiliskRun`, there is no guarantee that `fun` has access to the environment in which `basiliskRun` is called. This has a number of consequences for the type of code that can be written inside `fun`:

- Functions or variables from non-base R packages used inside `fun` should be prefixed with the package namespace, or the package itself should be reloaded inside `fun`.
- Any other variables used inside `fun` should be explicitly passed as an argument. Developers should not rely on closures to capture variables in the calling environment of `basiliskRun`.
- Relevant global variables should be reset inside `fun`.
- 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.

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`. The motivation for this is to avoid portability problems with hard-coded paths when **basilisk** is provided as a binary.

By default, both the base conda installation and the environments will be placed in an external user-writable directory. 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 installer.

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

Author(s)

Aaron Lun

See Also

`setupBasiliskEnv`, to set up the conda environments.

`getBasiliskFork` and `getBasiliskShared`, to control various global options.

Examples

```
# Loading one environment:
tmploc <- file.path(tempdir(), "my_package_B")
setupBasiliskEnv(tmploc, c('pandas=0.25.1',
    "python-dateutil=2.8.0", "pytz=2019.3"))

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

# Co-exists with our other environment:
tmploc2 <- file.path(tempdir(), "my_package_C")
setupBasiliskEnv(tmploc2, c('pandas=0.24.1',
    "python-dateutil=2.7.1", "pytz=2018.7"))

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() {
    assign(x="snake.in.my.shoes", 1, envir=basilisk::findPersistentEnv())
})
basiliskRun(proc=cl, function() {
    get("snake.in.my.shoes", envir=basilisk::findPersistentEnv())
})
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)
```

`findPersistentEnv` *Find the persistent environment*

Description

Find the persistent environment inside a `basiliskRun` call, to allow variables to be passed across calls.

Usage

```
findPersistentEnv()
```

Details

The persistent environment is where variables can be stored across `basiliskRun` calls. When `proc` is an environment, it serves as the persistent environment; otherwise, if `proc` is a process, the global environment of the process is the persistent environment.

Developers should avoid naming persistent variables with the `.basilisk` prefix. These are reserved for internal use and may be overwritten by later calls to `basiliskRun`.

Value

An environment to which persistent variables can be assigned, for use in later `basiliskRun` calls on the same `proc`.

Author(s)

Aaron Lun

See Also

`basiliskRun`, where this function can be used.

Examples

```
# Using the base environment for brevity.
cl <- basiliskStart(NULL)
basiliskRun(proc=cl, function() {
    assign(x="snake.in.my.shoes", 1, envir=basilisk::findPersistentEnv())
})
basiliskRun(proc=cl, function() {
    get("snake.in.my.shoes", envir=basilisk::findPersistentEnv())
})
basiliskStop(cl)
```

getBasiliskFork *Options for* **basilisk**

Description

Options controlling the efficiency and friendliness of starting up a Python instance via **basilisk**.

Usage

```
getBasiliskFork()

setBasiliskFork(value)

getBasiliskShared()

setBasiliskShared(value)
```

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.

Details

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.

Value

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

Author(s)

Aaron Lun

See Also

`basiliskStart`, where these options are used.

Examples

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

`listCorePackages` *List core packages*

Description

List the set of core Python packages (and their version numbers) that are provided by **basilisk**.

Usage

```
listCorePackages()
```

Details

The composition of the core list is determined by the Miniconda list for Python 3.7 (`https://docs.anaconda.com/anaconda/packages/pkg-docs/`). Note that there are subtle differences between the package lists for different operating systems; developers of clients of **basilisk** should avoid such OS-specific core packages.

Value

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

Author(s)

Aaron Lun

Examples

```
listCorePackages()
```

`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 a* **basilisk** *environments*

Description

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

Usage

```
setupBasiliskEnv(envpath, packages, pip = 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.
`pip`	Character vector containing the names of additional packages to install from PyPi using pip. Version numbers must be included.

Details

basilisk environments are simply Python conda environments that are created and managed by **basilisk**. Each **basilisk** environment can contain different Python packages with different versions, allowing us to avoid version conflicts within an R session when different Bioconductor packages (or even different functions within a single package) require incompatible versions of Python packages.

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` is set.

If a **basilisk** environment is already present at `envpath`, `setupBasiliskEnv` is a no-op. This ensures that the function only installs the packages once.

Value

A conda environment is created at `envpath` containing the specified `packages`. The function will return a logical scalar indicating whether creation was performed, which will be `FALSE` if the environment already exists.

Versioning

Pinned version numbers must be present for all requested conda packages in `packages`. This improved predictability makes debugging much easier when the R package is installed and executed on different systems. Note that this refers to conda packages, not Python packages, where the version notation for the former uses a single =; any == will be coerced to = 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 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 dependency resolver defaults to a different version of a required package. We suggest using `conda env export` to identify relevant dependencies and include them in `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 `conda env export`.

It is possible to specify a different version of Python in `packages` by supplying, e.g., `"python=2.7.10"`. If no Python version is listed, the version in the base conda installation is used by default.

See Also

`listCorePackages`, for a list of core Python packages with pinned versions.

Examples

```
tmploc <- file.path(tempdir(), "my_package_A")
setupBasiliskEnv(tmploc, c('pandas=0.25.3',
    "python-dateutil=2.8.1", "pytz=2019.3"))
```

useBasiliskEnv — *Use* **basilisk** *environments*

Description

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

Usage

```
useBasiliskEnv(envpath, dry = FALSE, required = TRUE)
```

Arguments

`envpath`	String containing the path to the **basilisk** environment to use.
`dry`	Logical scalar indicating whether only the directory should be returned without loading the environment.
`required`	Logical scalar indicating whether an error should be raised if the requested environment cannot be found.

Details

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

Direct use of this function with `dry=FALSE` will modify some environment variables for the current R session:

- The `"PYTHONPATH"` environment variable is unset. This is a deliberate choice to avoid compromising the version guarantees if `import` is allowed to search other locations beyond the specified **basilisk** environment.
- The `"RETICULATE_PYTHON"` environment variable is unset. This would otherwise override any choice of Python, including explicit specification via `use_condaenv`!
- Certain conda-related variables are modified to mimic activation of the desired conda environment in `envpath`. Actual activation seems to require modification of various files (e.g., `.bashrc`) which is undesirable.

If `dry=TRUE`, no environment variables are modified. Similarly, if the loading of the environment in `envpath` was not successful (i.e., `loaded` is `FALSE`), no environment variables are modified and `previous` is an empty list. Further note that `basiliskStop` will restore these environment variables to their state prior to running `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).

It returns a list containing:

- `loaded`, a logical scalar indicating whether the Python version in `envpath` was correctly loaded.
- `previous`, a list of environment variables with their values prior to running this function.

If `dry=TRUE`, only the logical scalar in `loaded` is returned directly.

Author(s)

Aaron Lun

See Also

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

Examples

```
# This may return TRUE or FALSE, depending on the available Python.
tmploc <- file.path(tempdir(), "my_package_B")
setupBasiliskEnv(tmploc, c('pandas==0.25.1',
    "python-dateutil=2.8.0", "pytz=2019.3"))
useBasiliskEnv(tmploc, required=FALSE)

# This will return FALSE, as the available Python is already set.
baseloc <- basilisk.utils::getBasiliskDir()
useBasiliskEnv(baseloc, required=FALSE)
```

Index