Package: reticulate (via r-universe)

June 28, 2024

Type Package

Title Interface to 'Python'

Version 1.38.0.9000

Description Interface to 'Python' modules, classes, and functions. When calling into 'Python', R data types are automatically converted to their equivalent 'Python' types. When values are returned from 'Python' to R they are converted back to R types. Compatible with all versions of 'Python' >= 2.7.

License Apache License 2.0

URL https://rstudio.github.io/reticulate/, https://github.com/rstudio/reticulate

BugReports https://github.com/rstudio/reticulate/issues

SystemRequirements Python (>= 2.7.0)

Encoding UTF-8

Depends R (>= 3.5)

Imports Matrix, Rcpp (>= 1.0.7), RcppTOML, graphics, here, jsonlite, methods, png, rappdirs, utils, rlang, withr

Suggests callr, knitr, glue, cli, rmarkdown, pillar, testthat

LinkingTo Rcpp

RoxygenNote 7.3.1

Roxygen list(markdown = TRUE)

VignetteBuilder knitr

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

RemoteUrl https://github.com/rstudio/reticulate

RemoteRef HEAD

RemoteSha 740169ae2a7c943115b2a4eca3b5a0b7d9e5ef3a

Contents

`==.python.builtin.object`

S3 Ops Methods for Python Objects

Description

Reticulate provides S3 Ops Group Generic Methods for Python objects. The methods invoke the equivalent python method of the object.

Usage

```
## S3 method for class 'python.builtin.object'
e1 == e2

## S3 method for class 'python.builtin.object'
e1 != e2

## S3 method for class 'python.builtin.object'
e1 < e2

## S3 method for class 'python.builtin.object'
e1 > e2

## S3 method for class 'python.builtin.object'
e1 >= e2

## S3 method for class 'python.builtin.object'
e1 <= e2

## S3 method for class 'python.builtin.object'
e1 + e2
```

```
## S3 method for class 'python.builtin.object'
e1 - e2

## S3 method for class 'python.builtin.object'
e1 * e2

## S3 method for class 'python.builtin.object'
e1 / e2

## S3 method for class 'python.builtin.object'
e1 %/% e2

## S3 method for class 'python.builtin.object'
e1 %% e2

## S3 method for class 'python.builtin.object'
e1 ^ e2

## S3 method for class 'python.builtin.object'
e1 & e2

## S3 method for class 'python.builtin.object'
e1 | e2

## S3 method for class 'python.builtin.object'
!e1

## S3 method for class 'python.builtin.object'
x %*% y
```

Arguments

e1, e2, x, y	A python object.

Value

Result from evaluating the Python expression. If either of the arguments to the operator was a Python object with `convert=FALSE`, then the result will also be a Python object with `convert=FALSE` set. Otherwise, the result will be converted to an R object if possible.

Operator Mappings

R expression	Python expression	First python method invoked
`x == y`	`x == y`	`type(x).__eq__(x, y)`
`x != y`	`x != y`	`type(x).__ne__(x, y)`
`x < y`	`x < y`	`type(x).__lt__(x, y)`
`x > y`	`x > y`	`type(x).__gt__(x, y)`
`x >= y`	`x >= y`	`type(x).__ge__(x, y)`

`x <= y`	`x <= y`	`type(x).__le__(x, y)`
`+ x`	`+ x`	`type(x).__pos__(x)`
`- y`	`- x`	`type(x).__neg__(x)`
`x + y`	`x + y`	`type(x).__add__(x, y)`
`x - y`	`x - y`	`type(x).__sub__(x, y)`
`x * y`	`x * y`	`type(x).__mul__(x, y)`
`x / y`	`x / y`	`type(x).__truediv__(x, y)`
`x %/% y`	`x // y`	`type(x).__floordiv__(x, y)`
`x %% y`	`x % y`	`type(x).__mod__(x, y)`
`x ^ y`	`x ** y`	`type(x).__pow__(x, y)`
`x & y`	`x & y`	`type(x).__and__(x, y)`
`x | y`	`x | y`	`type(x).__or__(x, y)`
`!x`	`~x`	`type(x).__not__(x)`
`x %*% y`	`x @ y`	`type(x).__matmul__(x, y)`

Note: If the initial Python method invoked raises a `NotImplemented` Exception, the Python interpreter will attempt to use the reflected variant of the method from the second argument. The arithmetic operators will call the equivalent double underscore (dunder) method with an "r" prefix. For instance, when evaluating the expression `x + y`, if `type(x).__add__(x, y)` raises a `NotImplemented` exception, then the interpreter will attempt `type(y).__radd__(y, x)`. The comparison operators follow a different sequence of fallbacks; refer to the Python documentation for more details.

`array_reshape` *Reshape an Array*

Description

Reshape (reindex) a multi-dimensional array, using row-major (C-style) reshaping semantics by default.

Usage

```
array_reshape(x, dim, order = c("C", "F"))
```

Arguments

`x`	An array
`dim`	The new dimensions to be set on the array.
`order`	The order in which elements of `x` should be read during the rearrangement. `"C"` means elements should be read in row-major order, with the last index changing fastest; `"F"` means elements should be read in column-major order, with the first index changing fastest.

Details

This function differs from e.g. `dim(x) <- dim` in a very important way: by default, `array_reshape()` will fill the new dimensions in row-major (C-style) ordering, while `dim<-()` will fill new dimensions in column-major (Fortran-style) ordering. This is done to be consistent with libraries like NumPy, Keras, and TensorFlow, which default to this sort of ordering when reshaping arrays. See the examples for why this difference may be important.

Examples

```
## Not run:
# let's construct a 2x2 array from a vector of 4 elements
x <- 1:4

# rearrange will fill the array row-wise
array_reshape(x, c(2, 2))
#      [,1] [,2]
# [1,]    1    2
# [2,]    3    4
# setting the dimensions 'fills' the array col-wise
dim(x) <- c(2, 2)
x
#      [,1] [,2]
# [1,]    1    3
# [2,]    2    4

## End(Not run)
```

---

as.character.python.builtin.bytes

Convert Python bytes to an R character vector

Description

Convert Python bytes to an R character vector

Usage

```
## S3 method for class 'python.builtin.bytes'
as.character(x, encoding = "utf-8", errors = "strict", ...)
```

Arguments

`x`	object to be coerced or tested.
`encoding`	Encoding to use for conversion (defaults to utf-8)
`errors`	Policy for handling conversion errors. Default is 'strict' which raises an error. Other possible values are 'ignore' and 'replace'.
`...`	further arguments passed to or from other methods.

`as_iterator` *Traverse a Python iterator or generator*

Description

Traverse a Python iterator or generator

Usage

```
as_iterator(x)

iterate(it, f = base::identity, simplify = TRUE)

iter_next(it, completed = NULL)
```

Arguments

`x`	Python iterator or iterable
`it`	Python iterator or generator
`f`	Function to apply to each item. By default applies the `identity` function which just reflects back the value of the item.
`simplify`	Should the result be simplified to a vector if possible?
`completed`	Sentinel value to return from `iter_next()` if the iteration completes (defaults to `NULL` but can be any R value you specify).

Details

Simplification is only attempted all elements are length 1 vectors of type "character", "complex", "double", "integer", or "logical".

Value

For `iterate()`, A list or vector containing the results of calling `f` on each item in `x` (invisibly); For `iter_next()`, the next value in the iteration (or the sentinel `completed` value if the iteration is complete).

conda-tools *Conda Tools*

Description

Tools for managing Python conda environments.

Usage

```
conda_list(conda = "auto")

conda_create(
  envname = NULL,
  packages = NULL,
  ...,
  forge = TRUE,
  channel = character(),
  environment = NULL,
  conda = "auto",
  python_version = miniconda_python_version(),
  additional_create_args = character()
)

conda_clone(envname, ..., clone = "base", conda = "auto")

conda_export(
  envname,
  file = if (json) "environment.json" else "environment.yml",
  json = FALSE,
  ...,
  conda = "auto"
)

conda_remove(envname, packages = NULL, conda = "auto")

conda_install(
  envname = NULL,
  packages,
  forge = TRUE,
  channel = character(),
  pip = FALSE,
  pip_options = character(),
  pip_ignore_installed = FALSE,
  conda = "auto",
  python_version = NULL,
  additional_create_args = character(),
  additional_install_args = character(),
```

```
  ...
)

conda_binary(conda = "auto")

conda_exe(conda = "auto")

conda_version(conda = "auto")

conda_update(conda = "auto")

conda_python(envname = NULL, conda = "auto", all = FALSE)

conda_search(
  matchspec,
  forge = TRUE,
  channel = character(),
  conda = "auto",
  ...
)

condaenv_exists(envname = NULL, conda = "auto")
```

Arguments

conda	The path to a conda executable. Use "auto" to allow reticulate to automatically find an appropriate conda binary. See **Finding Conda** and conda_binary() for more details.
envname	The name of, or path to, a conda environment.
packages	A character vector, indicating package names which should be installed or removed. Use <package>==<version> to request the installation of a specific version of a package. A NULL value for conda_remove() will be interpretted to "--all", removing the entire environment.
...	Optional arguments, reserved for future expansion.
forge	Boolean; include the conda-forge repository?
channel	An optional character vector of conda channels to include. When specified, the forge argument is ignored. If you need to specify multiple channels, including the conda forge, you can use c("conda-forge", <other channels>).
environment	The path to an environment definition, generated via (for example) conda_export(), or via conda env export. When provided, the conda environment will be created using this environment definition, and other arguments will be ignored.
python_version	The version of Python to be installed. Set this if you'd like to change the version of Python associated with a particular conda environment.
additional_create_args	An optional character vector of additional arguments to use in the call to conda create.
clone	The name of the conda environment to be cloned.

file	The path where the conda environment definition will be written.
json	Boolean; should the environment definition be written as JSON? By default, conda exports environments as YAML.
pip	Boolean; use pip for package installation? By default, packages are installed from the active conda channels.
pip_options	An optional character vector of additional command line arguments to be passed to pip. Only relevant when pip = TRUE.
pip_ignore_installed	Ignore already-installed versions when using pip? (defaults to FALSE). Set this to TRUE so that specific package versions can be installed even if they are downgrades. The FALSE option is useful for situations where you don't want a pip install to attempt an overwrite of a conda binary package (e.g. SciPy on Windows which is very difficult to install via pip due to compilation requirements).
additional_install_args	An optional character vector of additional arguments to use in the call to conda install.
all	Boolean; report all instances of Python found?
matchspec	A conda MatchSpec query string.

Value

conda_list() returns an R data.frame, with name giving the name of the associated environment, and python giving the path to the Python binary associated with that environment.

conda_create() returns the path to the Python binary associated with the newly-created conda environment.

conda_clone() returns the path to Python within the newly-created conda environment.

conda_export() returns the path to the exported environment definition, invisibly.

conda_search() returns an R data.frame describing packages that matched against matchspec. The data frame will usually include fields name giving the package name, version giving the package version, build giving the package build, and channel giving the channel the package is hosted on.

Finding Conda

Most of reticulate's conda APIs accept a conda parameter, used to control the conda binary used in their operation. When conda = "auto", reticulate will attempt to automatically find a conda installation. The following locations are searched, in order:

1. The location specified by the reticulate.conda_binary R option,
2. The location specified by the RETICULATE_CONDA environment variable,
3. The miniconda_path() location (if it exists),
4. The program PATH,
5. A set of pre-defined locations where conda is typically installed.

To force reticulate to use a particular conda binary, we recommend setting:

```
options(reticulate.conda_binary = "/path/to/conda")
```

This can be useful if your conda installation lives in a location that `reticulate` is unable to automatically discover.

`configure_environment` *Configure a Python Environment*

Description

Configure a Python environment, satisfying the Python dependencies of any loaded R packages.

Usage

```
configure_environment(package = NULL, force = FALSE)
```

Arguments

`package`	The name of a package to configure. When `NULL`, `reticulate` will instead look at all loaded packages and discover their associated Python requirements.
`force`	Boolean; force configuration of the Python environment? Note that `configure_environment()` is a no-op within non-interactive R sessions. Use this if you require automatic environment configuration, e.g. when testing a package on a continuous integration service.

Details

Normally, this function should only be used by package authors, who want to ensure that their package dependencies are installed in the active Python environment. For example:

```
.onLoad <- function(libname, pkgname) {
  reticulate::configure_environment(pkgname)
}
```

If the Python session has not yet been initialized, or if the user is not using the default Miniconda Python installation, no action will be taken. Otherwise, `reticulate` will take this as a signal to install any required Python dependencies into the user's Python environment.

If you'd like to disable `reticulate`'s auto-configure behavior altogether, you can set the environment variable:

```
RETICULATE_AUTOCONFIGURE = FALSE
```

e.g. in your `~/.Renviron` or similar.

Note that, in the case where the Python session has not yet been initialized, `reticulate` will automatically ensure your required Python dependencies are installed after the Python session is initialized (when appropriate).

dict *Create Python dictionary*

Description

Create a Python dictionary object, including a dictionary whose keys are other Python objects rather than character vectors.

Usage

```
dict(..., convert = FALSE)

py_dict(keys, values, convert = FALSE)
```

Arguments

...	Name/value pairs for dictionary (or a single named list to be converted to a dictionary).
convert	TRUE to automatically convert Python objects to their R equivalent. If you pass FALSE you can do manual conversion using the py_to_r() function.
keys	Keys to dictionary (can be Python objects)
values	Values for dictionary

Value

A Python dictionary

Note

The returned dictionary will not automatically convert its elements from Python to R. You can do manual conversion with the py_to_r() function or pass convert = TRUE to request automatic conversion.

eng_python *A reticulate Engine for Knitr*

Description

This provides a reticulate engine for knitr, suitable for usage when attempting to render Python chunks. Using this engine allows for shared state between Python chunks in a document – that is, variables defined by one Python chunk can be used by later Python chunks.

Usage

```
eng_python(options)
```

Arguments

options	Chunk options, as provided by knitr during chunk execution.

Details

The engine can be activated by setting (for example)

```
knitr::knit_engines$set(python = reticulate::eng_python)
```

Typically, this will be set within a document's setup chunk, or by the environment requesting that Python chunks be processed by this engine. Note that knitr (since version 1.18) will use the reticulate engine by default when executing Python chunks within an R Markdown document.

Supported knitr chunk options

For most options, reticulate's python engine behaves the same as the default R engine included in knitr, but they might not support all the same features. Options in *italic* are equivalent to knitr, but with modified behavior.

- eval (TRUE, logical): If TRUE, all expressions in the chunk are evaluated. If FALSE, no expression is evaluated. Unlike knitr's R engine, it doesn't support numeric values indicating the expressions to evaluate.
- echo (TRUE, logical): Whether to display the source code in the output document. Unlike knitr's R engine, it doesn't support numeric values indicating the expressions to display.
- results ('markup', character): Controls how to display the text results. Note that this option only applies to normal text output (not warnings, messages, or errors). The behavior should be identical to knitr's R engine.
- collapse (FALSE, logical): Whether to, if possible, collapse all the source and output blocks from one code chunk into a single block (by default, they are written to separate blocks). This option only applies to Markdown documents.
- error (TRUE, logical): Whether to preserve errors. If FALSE evaluation stops on errors. (Note that RMarkdown sets it to FALSE).
- warning (TRUE, logical): Whether to preserve warnings in the output. If FALSE, all warnings will be suppressed. Doesn't support indices.
- include (TRUE, logical): Whether to include the chunk output in the output document. If FALSE, nothing will be written into the output document, but the code is still evaluated and plot files are generated if there are any plots in the chunk, so you can manually insert figures later.
- dev: The graphical device to generate plot files. See knitr documentation for additional information.
- base.dir (NULL; character): An absolute directory under which the plots are generated.
- strip.white (TRUE; logical): Whether to remove blank lines in the beginning or end of a source code block in the output.
- dpi (72; numeric): The DPI (dots per inch) for bitmap devices (dpi * inches = pixels).
- fig.width, fig.height (both are 7; numeric): Width and height of the plot (in inches), to be used in the graphics device.

- `label`: The chunk label for each chunk is assumed to be unique within the document. This is especially important for cache and plot filenames, because these filenames are based on chunk labels. Chunks without labels will be assigned labels like unnamed-chunk-i, where i is an incremental number.

Python engine only options:

- `jupyter_compat` (FALSE, logical): If `TRUE` then, like in Jupyter notebooks, only the last expression in the chunk is printed to the output.
- `out.width.px`, `out.height.px` (810, 400, both integers): Width and height of the plot in the output document, which can be different with its physical `fig.width` and `fig.height`, i.e., plots can be scaled in the output document. Unlike knitr's `out.width`, this is always set in pixels.
- `altair.fig.width`, `altair.fig.height`: If set, is used instead of `out.width.px` and `out.height.px` when writing Altair charts.

`import` *Import a Python module*

Description

Import the specified Python module, making it available for use from R.

Usage

```
import(module, as = NULL, convert = TRUE, delay_load = FALSE)

import_main(convert = TRUE, delay_load = FALSE)

import_builtins(convert = TRUE, delay_load = FALSE)

import_from_path(module, path = ".", convert = TRUE, delay_load = FALSE)
```

Arguments

`module`	The name of the Python module.
`as`	An alias for module name (affects names of R classes). Note that this is an advanced parameter that should generally only be used in package development (since it affects the S3 name of the imported class and can therefore interfere with S3 method dispatching).
`convert`	Boolean; should Python objects be automatically converted to their R equivalent? If set to `FALSE`, you can still manually convert Python objects to R via the `py_to_r()` function.
`delay_load`	Boolean; delay loading the module until it is first used? When `FALSE`, the module will be loaded immediately. See **Delay Load** for advanced usages.
`path`	The path from which the module should be imported.

Value

An R object wrapping a Python module. Module attributes can be accessed via the $ operator, or via py_get_attr().

Python Built-ins

Python's built-in functions (e.g. len()) can be accessed via Python's built-in module. Because the name of this module has changed between Python 2 and Python 3, we provide the function import_builtins() to abstract over that name change.

Delay Load

The delay_load parameter accepts a variety of inputs. If you just need to ensure your module is lazy-loaded (e.g. because you are a package author and want to avoid initializing Python before the user has explicitly requested it), then passing TRUE is normally the right choice.

You can also provide a named list: "before_load", "on_load" and "on_error" can be functions , which act as callbacks to be run when the module is later loaded. "environment" can be a character vector of preferred python environment names to search for and use. For example:

```
delay_load = list(

  # run before the module is loaded
  before_load = function() { ... }

  # run immediately after the module is loaded
  on_load = function() { ... }

  # run if an error occurs during module import
  on_error = function(error) { ... }

  environment = c("r-preferred-venv1", "r-preferred-venv2")
)
```

Alternatively, if you supply only a single function, that will be treated as an on_load handler.

Import from Path

import_from_path() can be used in you need to import a module from an arbitrary filesystem path. This is most commonly used when importing modules bundled with an R package – for example:

```
path <- system.file("python", package = <package>)
reticulate::import_from_path(<module>, path = path, delay_load = TRUE)
```

Examples

```
## Not run:
main <- import_main()
```

```
sys <- import("sys")

## End(Not run)
```

`install_miniconda` *Install Miniconda*

Description

Download the Miniconda installer, and use it to install Miniconda.

Usage

```
install_miniconda(path = miniconda_path(), update = TRUE, force = FALSE)
```

Arguments

`path`	The location where Miniconda is (or should be) installed. Note that the Miniconda installer does not support paths containing spaces. See miniconda_path for more details on the default path used by `reticulate`.
`update`	Boolean; update to the latest version of Miniconda after installation?
`force`	Boolean; force re-installation if Miniconda is already installed at the requested path?

Details

For arm64 builds of R on macOS, `install_miniconda()` will use binaries from miniforge instead.

Note

If you encounter binary incompatibilities between R and Miniconda, a scripted build and installation of Python from sources can be performed by `install_python()`

See Also

Other miniconda-tools: `miniconda_uninstall()`, `miniconda_update()`

install_python *Install Python*

Description

Download and install Python, using the pyenv. and pyenv-win projects.

Usage

```
install_python(
  version = "3.10:latest",
  list = FALSE,
  force = FALSE,
  optimized = TRUE
)
```

Arguments

version	The version of Python to install.
list	Boolean; if set, list the set of available Python versions?
force	Boolean; force re-installation even if the requested version of Python is already installed?
optimized	Boolean; if TRUE, installation will take significantly longer but should result in a faster Python interpreter. Only applicable on macOS and Linux.

Details

In general, it is recommended that Python virtual environments are created using the copies of Python installed by install_python(). For example:

```
library(reticulate)
version <- "3.9.12"
install_python(version)
virtualenv_create("my-environment", version = version)
use_virtualenv("my-environment")

# There is also support for a ":latest" suffix to select the latest patch release
install_python("3.9:latest") # install latest patch available at python.org

# select the latest 3.9.* patch installed locally
virtualenv_create("my-environment", version = "3.9:latest")
```

Note

On macOS and Linux this will build Python from sources, which may take a few minutes. Installation will be faster if some build dependencies are preinstalled. See https://github.com/pyenv/pyenv/wiki#suggested-build-environment for example commands you can run to pre-install system dependencies (requires administrator privileges).

If `optimized = TRUE`, (the default) Python is build with:

```
PYTHON_CONFIGURE_OPTS="--enable-shared --enable-optimizations --with-lto"
PYTHON_CFLAGS="-march=native -mtune=native"
```

If `optimized = FALSE`, Python is built with:

```
PYTHON_CONFIGURE_OPTS=--enable-shared
```

On Windows, prebuilt installers from https://www.python.org are used.

`miniconda_path` *Path to Miniconda*

Description

The path to the Miniconda installation to use. By default, an OS-specific path is used. If you'd like to instead set your own path, you can set the `RETICULATE_MINICONDA_PATH` environment variable.

Usage

```
miniconda_path()
```

`miniconda_uninstall` *Remove Miniconda*

Description

Uninstall Miniconda.

Usage

```
miniconda_uninstall(path = miniconda_path())
```

Arguments

`path`	The path in which Miniconda is installed.

See Also

Other miniconda-tools: `install_miniconda()`, `miniconda_update()`

miniconda_update *Update Miniconda*

Description

Update Miniconda to the latest version.

Usage

```
miniconda_update(path = miniconda_path())
```

Arguments

path	The location where Miniconda is (or should be) installed. Note that the Miniconda installer does not support paths containing spaces. See miniconda_path for more details on the default path used by reticulate.

See Also

Other miniconda-tools: install_miniconda(), miniconda_uninstall()

nameOfClass.python.builtin.type *nameOfClass() for Python objects*

Description

This generic enables passing a python.builtin.type object as the 2nd argument to base::inherits().

Usage

```
## S3 method for class 'python.builtin.type'
nameOfClass(x)
```

Arguments

x	A Python class

Value

A scalar string matching the S3 class of objects constructed from the type.

Examples

```
## Not run:
  numpy <- import("numpy")
  x <- r_to_py(array(1:3))
  inherits(x, numpy$ndarray)

## End(Not run)
```

---

np_array *NumPy array*

Description

Create NumPy arrays and convert the data type and in-memory ordering of existing NumPy arrays.

Usage

```
np_array(data, dtype = NULL, order = "C")
```

Arguments

data	Vector or existing NumPy array providing data for the array
dtype	Numpy data type (e.g. "float32", "float64", etc.)
order	Memory ordering for array. "C" means C order, "F" means Fortran order.

Value

A NumPy array object.

---

py *Interact with the Python Main Module*

Description

The py object provides a means for interacting with the Python main session directly from R. Python objects accessed through py are automatically converted into R objects, and can be used with any other R functions as needed.

Usage

```
py
```

Format

An R object acting as an interface to the Python main module.

PyClass *Create a python class*

Description

Create a python class

Usage

```
PyClass(classname, defs = list(), inherit = NULL)
```

Arguments

classname	Name of the class. The class name is useful for S3 method dispatch.
defs	A named list of class definitions - functions, attributes, etc.
inherit	A list of Python class objects. Usually these objects have the python.builtin.type S3 class.

Examples

```
## Not run:
Hi <- PyClass("Hi", list(
  name = NULL,
  `__init__` = function(self, name) {
    self$name <- name
    NULL
  },
  say_hi = function(self) {
    paste0("Hi ", self$name)
  }
))

a <- Hi("World")

## End(Not run)
```

py_available *Check if Python is available on this system*

Description

Check if Python is available on this system

Usage

```
py_available(initialize = FALSE)

py_numpy_available(initialize = FALSE)
```

Arguments

initialize	TRUE to attempt to initialize Python bindings if they aren't yet available (defaults to FALSE).

Value

Logical indicating whether Python is initialized.

Note

The py_numpy_available function is a superset of the py_available function (it calls py_available first before checking for NumPy).

---

py_bool *Python Truthiness*

Description

Equivalent to bool(x) in Python, or not not x.

Usage

```
py_bool(x)
```

Arguments

x	A python object.

Details

If the Python object defines a __bool__ method, then that is invoked. Otherwise, if the object defines a __len__ method, then TRUE is returned if the length is nonzero. If neither __len__ nor __bool__ are defined, then the Python object is considered TRUE.

Value

An R scalar logical: TRUE or FALSE. If x is a null pointer or Python is not initialized, FALSE is returned.

py_capture_output *Capture and return Python output*

Description

Capture and return Python output

Usage

```
py_capture_output(expr, type = c("stdout", "stderr"))
```

Arguments

expr	Expression to capture stdout for
type	Streams to capture (defaults to both stdout and stderr)

Value

Character vector with output

py_clear_last_error *Get or (re)set the last Python error encountered.*

Description

Get or (re)set the last Python error encountered.

Usage

```
py_clear_last_error()

py_last_error(exception)
```

Arguments

exception	A python exception object. If provided, the provided exception is set as the last exception.

Value

For `py_last_error()`, `NULL` if no error has yet been encountered. Otherwise, a named list with entries:

- `"type"`: R string, name of the exception class.
- `"value"`: R string, formatted exception message.
- `"traceback"`: R character vector, the formatted python traceback,
- `"message"`: The full formatted raised exception, as it would be printed in Python. Includes the traceback, type, and value.
- `"r_trace"`: A `data.frame` with class `rlang_trace` and columns:
  - `call`: The R callstack, `full_call`, summarized for pretty printing.
  - `full_call`: The R callstack. (Output of `sys.calls()` at the error callsite).
  - `parent`: The parent of each frame in callstack. (Output of `sys.parents()` at the error callsite).
  - Additional columns for internals use: `namespace`, `visible`, `scope`.

And attribute `"exception"`, a `'python.builtin.Exception'` object.

The named list has `class "py_error"`, and has a default `print` method that is the equivalent of `cat(py_last_error()$message)`.

Examples

```
## Not run:

# see last python exception with R traceback
reticulate::py_last_error()

# see the full R callstack from the last Python exception
reticulate::py_last_error()$r_trace$full_call

# run python code that might error,
# without modifying the user-visible python exception

safe_len <- function(x) {
  last_err <- py_last_error()
  tryCatch({
    # this might raise a python exception if x has no `__len__` method.
    import_builtins()$len(x)
  }, error = function(e) {
    # py_last_error() was overwritten, is now "no len method for 'object'"
    py_last_error(last_err) # restore previous exception
    -1L
  })
}

safe_len(py_eval("object"))

## End(Not run)
```

py_config *Python configuration*

Description

Retrieve information about the version of Python currently being used by `reticulate`.

Usage

```
py_config()
```

Details

If Python has not yet been initialized, then calling `py_config()` will force the initialization of Python. See `py_discover_config()` for more details.

Value

Information about the version of Python in use, as an R list with class `"py_config"`.

py_del_attr *Delete an attribute of a Python object*

Description

Delete an attribute of a Python object

Usage

```
py_del_attr(x, name)
```

Arguments

`x`	A Python object.
`name`	The attribute name.

py_discover_config *Discover the version of Python to use with reticulate.*

Description

This function enables callers to check which versions of Python will be discovered on a system as well as which one will be chosen for use with reticulate.

Usage

```
py_discover_config(required_module = NULL, use_environment = NULL)
```

Arguments

required_module	A optional module name that will be used to select the Python environment used.
use_environment	An optional virtual/conda environment name to prefer in the search.

Details

The order of discovery is documented in `vignette("versions")`, also available online here

Value

Python configuration object.

py_ellipsis *The builtin constant Ellipsis*

Description

The builtin constant Ellipsis

Usage

```
py_ellipsis()
```

`py_eval` *Evaluate a Python Expression*

Description

Evaluate a single Python expression, in a way analogous to the Python `eval()` built-in function.

Usage

```
py_eval(code, convert = TRUE)
```

Arguments

`code`	A single Python expression.
`convert`	Boolean; automatically convert Python objects to R?

Value

The result produced by evaluating `code`, converted to an `R` object when `convert` is set to `TRUE`.

Caveats

`py_eval()` only supports evaluation of 'simple' Python expressions. Other expressions (e.g. assignments) will fail; e.g.

```
> py_eval("x = 1")
Error in py_eval_impl(code, convert) :
  SyntaxError: invalid syntax (reticulate_eval, line 1)
```

and this mirrors what one would see in a regular Python interpreter:

```
>>> eval("x = 1")
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
  File "<string>", line 1
x = 1
^
  SyntaxError: invalid syntax
```

The `py_run_string()` method can be used if the evaluation of arbitrary Python code is required.

py_exe *Python executable*

Description

Get the path to the Python executable that `reticulate` has been configured to use. If Python has already been initialized, then `reticulate` will choose the currently-active copy of Python.

Usage

```
py_exe()
```

Details

This can occasionally be useful if you'd like to interact with Python (or its modules) via a subprocess; for example you might choose to install a package with `pip`:

```
system2(py_exe(), c("-m", "pip", "install", "numpy"))
```

and so you can also have greater control over how these modules are invoked.

Value

The path to the Python executable `reticulate` has been configured to use.

py_func *Wrap an R function in a Python function with the same signature.*

Description

This function could wrap an R function in a Python function with the same signature. Note that the signature of the R function must not contain esoteric Python-incompatible constructs.

Usage

```
py_func(f)
```

Arguments

`f`	An R function

Value

A Python function that calls the R function `f` with the same signature.

py_function_custom_scaffold *Custom Scaffolding of R Wrappers for Python Functions*

Description

This function can be used to generate R wrapper for a specified Python function while allowing to inject custom code for critical parts of the wrapper generation, such as process the any part of the docs obtained from `py_function_docs()` and append additional roxygen fields. The result from execution of `python_function` is assigned to a variable called `python_function_result` that can also be processed by `postprocess_fn` before writing the closing curly braces for the generated wrapper function.

Usage

```
py_function_custom_scaffold(
  python_function,
  r_function = NULL,
  additional_roxygen_fields = NULL,
  process_docs_fn = function(docs) docs,
  process_param_fn = function(param, docs) param,
  process_param_doc_fn = function(param_doc, docs) param_doc,
  postprocess_fn = function() {
 },
  file_name = NULL
)
```

Arguments

`python_function`	Fully qualified name of Python function or class constructor (e.g. `tf$layers$average_pooling1d`)
`r_function`	Name of R function to generate (defaults to name of Python function if not specified)
`additional_roxygen_fields`	A list of additional roxygen fields to write to the roxygen docs, e.g. `list(export = "", rdname = "generated-wrappers")`.
`process_docs_fn`	A function to process docs obtained from `reticulate::py_function_docs(python_function)`.
`process_param_fn`	A function to process each parameter needed for `python_funcion` before executing `python_funcion`.
`process_param_doc_fn`	A function to process the roxygen docstring for each parameter.
`postprocess_fn`	A function to inject any custom code in the form of a string before writing the closing curly braces for the generated wrapper function.
`file_name`	The file name to write the generated wrapper function to. If `NULL`, the generated wrapper will only be printed out in the console.

Examples

```
## Not run:

library(tensorflow)
library(stringr)

# Example of a `process_param_fn` to cast parameters with default values
# that contains "L" to integers
process_int_param_fn <- function(param, docs) {
  # Extract the list of parameters that have integer values as default
  int_params <- gsub(
    " = [-]?[0-9]+L",
    "",
    str_extract_all(docs$signature, "[A-z]+ = [-]?[0-9]+L")[[1]])
  # Explicitly cast parameter in the list obtained above to integer
  if (param %in% int_params) {
    param <- paste0("as.integer(", param, ")")
  }
  param
}

# Note that since the default value of parameter `k` is `1L`. It is wrapped
# by `as.integer()` to ensure it's casted to integer before sending it to `tf$nn$top_k`
# for execution. We then print out the python function result.
py_function_custom_scaffold(
  "tf$nn$top_k",
  r_function = "top_k",
  process_param_fn = process_int_param_fn,
  postprocess_fn = function() { "print(python_function_result)" })


## End(Not run)
```

`py_get_attr` *Get an attribute of a Python object*

Description

Get an attribute of a Python object

Usage

```
py_get_attr(x, name, silent = FALSE)
```

Arguments

`x`	Python object
`name`	Attribute name
`silent`	`TRUE` to return `NULL` if the attribute doesn't exist (default is `FALSE` which will raise an error)

Value

Attribute of Python object

`py_get_item` *Get/Set/Delete an item from a Python object*

Description

Access an item from a Python object, similar to how `x[key]` might be used in Python code to access an item indexed by `key` on an object `x`. The object's `__getitem__()` `__setitem__()` or `__delitem__()` method will be called.

Usage

```
py_get_item(x, key, silent = FALSE)

py_set_item(x, key, value)

py_del_item(x, key)

## S3 method for class 'python.builtin.object'
x[...]

## S3 replacement method for class 'python.builtin.object'
x[...] <- value
```

Arguments

`x`	A Python object.
`key, ...`	The key used for item lookup.
`silent`	Boolean; when `TRUE`, attempts to access missing items will return `NULL` rather than throw an error.
`value`	The item value to set. Assigning `value` of `NULL` calls `py_del_item()` and is equivalent to the python expression `del x[key]`. To set an item value of `None`, you can call `py_set_item()` directly, or call `x[key] <- py_none()`

Value

For `py_get_item()` and `[`, the return value from the `x.__getitem__()` method. For `py_set_item()`, `py_del_item()` and `[<-`, the mutate object `x` is returned.

Note

The `py_get_item()` always returns an unconverted python object, while `[` will automatically attempt to convert the object if `x` was created with `convert = TRUE`.

Examples

```
## Not run:

## get/set/del item from Python dict
x <- r_to_py(list(abc = "xyz"))

#'   # R expression    | Python expression
# -------------------- | -----------------
 x["abc"]              # x["abc"]
 x["abc"] <- "123"     # x["abc"] = "123"
 x["abc"] <- NULL      # del x["abc"]
 x["abc"] <- py_none() # x["abc"] = None

## get item from Python list
x <- r_to_py(list("a", "b", "c"))
x[0]

## slice a NumPy array
x <- np_array(array(1:64, c(4, 4, 4)))

# R expression | Python expression
# ------------ | -----------------
  x[0]         # x[0]
  x[, 0]       # x[:, 0]
  x[, , 0]     # x[:, :, 0]

  x[NA:2]      # x[:2]
  x[`:2`]      # x[:2]

  x[2:NA]      # x[2:]
  x[`2:`]      # x[2:]

  x[NA:NA:2]   # x[::2]
  x[`::2`]     # x[::2]

  x[1:3:2]     # x[1:3:2]
  x[`1:3:2`]   # x[1:3:2]


## End(Not run)
```

---

py_has_attr *Check if a Python object has an attribute*

Description

Check whether a Python object x has an attribute name.

Usage

```
py_has_attr(x, name)
```

Arguments

x	A python object.
name	The attribute to be accessed.

Value

TRUE if the object has the attribute name, and FALSE otherwise.

---

py_help *Documentation for Python Objects*

Description

Documentation for Python Objects

Usage

```
py_help(object)
```

Arguments

object	Object to print documentation for

---

py_id *Unique identifer for Python object*

Description

Get a globally unique identifier for a Python object.

Usage

```
py_id(object)
```

Arguments

object	Python object

Value

Unique identifer (as string) or NULL

Note

In the current implementation of CPython this is the memory address of the object.

py_install *Install Python packages*

Description

Install Python packages into a virtual environment or Conda environment.

Usage

```
py_install(
  packages,
  envname = NULL,
  method = c("auto", "virtualenv", "conda"),
  conda = "auto",
  python_version = NULL,
  pip = FALSE,
  ...,
  pip_ignore_installed = ignore_installed,
  ignore_installed = FALSE
)
```

Arguments

packages	A vector of Python packages to install.
envname	The name, or full path, of the environment in which Python packages are to be installed. When NULL (the default), the active environment as set by the RETICULATE_PYTHON_ENV variable will be used; if that is unset, then the r-reticulate environment will be used.
method	Installation method. By default, "auto" automatically finds a method that will work in the local environment. Change the default to force a specific installation method. Note that the "virtualenv" method is not available on Windows.
conda	The path to a conda executable. Use "auto" to allow reticulate to automatically find an appropriate conda binary. See **Finding Conda** and conda_binary() for more details.
python_version	The requested Python version. Ignored when attempting to install with a Python virtual environment.
pip	Boolean; use pip for package installation? This is only relevant when Conda environments are used, as otherwise packages will be installed from the Conda repositories.
...	Additional arguments passed to conda_install() or virtualenv_install().
pip_ignore_installed, ignore_installed	Boolean; whether pip should ignore previously installed versions of the requested packages. Setting this to TRUE causes pip to install the latest versions of all dependencies into the requested environment. This ensure that no dependencies are satisfied by a package that exists either in the site library or was previously installed from a different–potentially incompatible–distribution channel.

(ignore_installed is an alias for pip_ignore_installed, pip_ignore_installed takes precedence).

Details

On Linux and OS X the "virtualenv" method will be used by default ("conda" will be used if virtualenv isn't available). On Windows, the "conda" method is always used.

See Also

conda_install(), for installing packages into conda environments. virtualenv_install(), for installing packages into virtual environments.

---

py_is_null_xptr *Check if a Python object is a null externalptr*

Description

Check if a Python object is a null externalptr

Usage

```
py_is_null_xptr(x)

py_validate_xptr(x)
```

Arguments

x	Python object

Details

When Python objects are serialized within a persisted R environment (e.g. .RData file) they are deserialized into null externalptr objects (since the Python session they were originally connected to no longer exists). This function allows you to safely check whether whether a Python object is a null externalptr.

The py_validate function is a convenience function which calls py_is_null_xptr and throws an error in the case that the xptr is NULL.

Value

Logical indicating whether the object is a null externalptr

`py_iterator` *Create a Python iterator from an R function*

Description

Create a Python iterator from an R function

Usage

```
py_iterator(fn, completed = NULL, prefetch = 0L)
```

Arguments

`fn`	R function with no arguments.
`completed`	Special sentinel return value which indicates that iteration is complete (defaults to `NULL`).
`prefetch`	Number items to prefetch. Set this to a positive integer to avoid a deadlock in situations where the generator values are consumed by python background threads while the main thread is blocked.

Details

Python generators are functions that implement the Python iterator protocol. In Python, values are returned using the `yield` keyword. In R, values are simply returned from the function.

In Python, the `yield` keyword enables successive iterations to use the state of previous iterations. In R, this can be done by returning a function that mutates its enclosing environment via the `<<-` operator. For example:

```
sequence_generator <- function(start) {
  value <- start
  function() {
    value <<- value + 1
    value
  }
}
```

Then create an iterator using `py_iterator()`:

```
g <- py_iterator(sequence_generator(10))
```

Value

Python iterator which calls the R function for each iteration.

Ending Iteration

In Python, returning from a function without calling `yield` indicates the end of the iteration. In R however, `return` is used to yield values, so the end of iteration is indicated by a special return value (`NULL` by default, however this can be changed using the `completed` parameter). For example:

```
sequence_generator <-function(start) {
  value <- start
  function() {
    value <<- value + 1
    if (value < 100)
      value
    else
      NULL
  }
}
```

Threading

Some Python APIs use generators to parallellize operations by calling the generator on a background thread and then consuming its results on the foreground thread. The `py_iterator()` function creates threadsafe iterators by ensuring that the R function is always called on the main thread (to be compatible with R's single-threaded runtime) even if the generator is run on a background thread.

`py_len` *Length of Python object*

Description

Get the length of a Python object. This is equivalent to calling the Python builtin `len()` function on the object.

Usage

```
py_len(x, default = NULL)
```

Arguments

`x`	A Python object.
`default`	The default length value to return, in the case that the associated Python object has no `__len__` method. When `NULL` (the default), an error is emitted instead.

Details

Not all Python objects have a defined length. For objects without a defined length, calling `py_len()` will throw an error. If you'd like to instead infer a default length in such cases, you can set the `default` argument to e.g. `1L`, to treat Python objects without a `__len__` method as having length one.

Value

The length of the object, as a numeric value.

---

py_list_attributes *List all attributes of a Python object*

Description

List all attributes of a Python object

Usage

```
py_list_attributes(x)
```

Arguments

x	Python object

Value

Character vector of attributes

---

py_list_packages *List installed Python packages*

Description

List the Python packages that are installed in the requested Python environment.

Usage

```
py_list_packages(
  envname = NULL,
  type = c("auto", "virtualenv", "conda"),
  python = NULL
)
```

Arguments

envname	The name of, or path to, a Python virtual environment. Ignored when python is non-NULL.
type	The virtual environment type. Useful if you have both virtual environments and Conda environments of the same name on your system, and you need to disambiguate them.
python	The path to a Python executable.

Details

When `envname` is `NULL`, `reticulate` will use the "default" version of Python, as reported by `py_exe()`. This implies that you can call `py_list_packages()` without arguments in order to list the installed Python packages in the version of Python currently used by `reticulate`.

Value

An R data.frame, with columns:

`package` The package name.

`version` The package version.

`requirement` The package requirement.

`channel` (Conda only) The channel associated with this package.

`py_main_thread_func` *Create a Python function that will always be called on the main thread*

Description

This function is helpful when you need to provide a callback to a Python library which may invoke the callback on a background thread. As R functions must run on the main thread, wrapping the R function with `py_main_thread_func()` will ensure that R code is only executed on the main thread.

Usage

```
py_main_thread_func(f)
```

Arguments

`f`	An R function with arbitrary arguments

Value

A Python function that delegates to the passed R function, which is guaranteed to always be called on the main thread.

py_module_available *Check if a Python module is available on this system.*

Description

Note that this function will also attempt to initialize Python before checking if the requested module is available.

Usage

```
py_module_available(module)
```

Arguments

module	The name of the module.

Value

TRUE if the module is available and can be loaded; FALSE otherwise.

py_none *The Python None object*

Description

Get a reference to the Python None object.

Usage

```
py_none()
```

py_repr *String representation of a python object.*

Description

This is equivalent to calling str(object) or repr(object) in Python.

Usage

```
py_repr(object)

py_str(object, ...)
```

Arguments

object	Python object
...	Unused

Details

In Python, calling `print()` invokes the builtin `str()`, while auto-printing an object at the REPL invokes the builtin `repr()`.

In R, the default print method for python objects invokes `py_repr()`, and the default `format()` and `as.character()` methods invoke `py_str()`.

For historical reasons, `py_str()` is also an R S3 method that allows R authors to customize the the string representation of a Python object from R. New code is recommended to provide a `format()` and/or `print()` S3 R method for python objects instead.

The default implementation will call `PyObject_Str` on the object.

Value

Character vector

---

py_run *Run Python code*

Description

Execute code within the scope of the `__main__` Python module.

Usage

```
py_run_string(code, local = FALSE, convert = TRUE)

py_run_file(file, local = FALSE, convert = TRUE, prepend_path = TRUE)
```

Arguments

code	The Python code to be executed.
local	Boolean; should Python objects be created as part of a local / private dictionary? If `FALSE`, objects will be created within the scope of the Python main module.
convert	Boolean; should Python objects be automatically converted to their R equivalent? If set to `FALSE`, you can still manually convert Python objects to R via the `py_to_r()` function.
file	The Python script to be executed.
prepend_path	Boolean; should the script directory be added to the Python module search path? The default, `TRUE`, matches the behavior of `python <path/to/script.py>` at the command line.

Value

A Python dictionary of objects. When `local` is `FALSE`, this dictionary captures the state of the Python main module after running the provided code. Otherwise, only the variables defined and used are captured.

`py_save_object` *Save and Load Python Objects*

Description

Save and load Python objects.

Usage

```
py_save_object(object, filename, pickle = "pickle", ...)

py_load_object(filename, pickle = "pickle", ..., convert = TRUE)
```

Arguments

`object`	A Python object.
`filename`	The output file name. Note that the file extension `.pickle` is considered the "standard" extension for serialized Python objects as created by the `pickle` module.
`pickle`	The "pickle" implementation to use. Defaults to `"pickle"`, but other compatible Python "pickle" implementations (e.g. `"cPickle"`) could be used as well.
`...`	Optional arguments, to be passed to the `pickle` module's `dump()` and `load()` functions.
`convert`	Bool. Whether the loaded pickle object should be converted to an R object.

Details

Python objects are serialized using the `pickle` module – see https://docs.python.org/3/library/pickle.html for more details.

py_set_attr *Set an attribute of a Python object*

Description

Set an attribute of a Python object

Usage

```
py_set_attr(x, name, value)
```

Arguments

x	Python object
name	Attribute name
value	Attribute value

py_set_seed *Set Python and NumPy random seeds*

Description

Set various random seeds required to ensure reproducible results. The provided seed value will establish a new random seed for Python and NumPy, and will also (by default) disable hash randomization.

Usage

```
py_set_seed(seed, disable_hash_randomization = TRUE)
```

Arguments

seed	A single value, interpreted as an integer
disable_hash_randomization	Disable hash randomization, which is another common source of variable results. See https://docs.python.org/3/using/cmdline.html#envvar-PYTHONHASHSEED

Details

This function does not set the R random seed, for that you should call set.seed().

py_suppress_warnings *Suppress Python warnings for an expression*

Description

Suppress Python warnings for an expression

Usage

```
py_suppress_warnings(expr)
```

Arguments

expr	Expression to suppress warnings for

Value

Result of evaluating expression

py_unicode *Convert to Python Unicode Object*

Description

Convert to Python Unicode Object

Usage

```
py_unicode(str)
```

Arguments

str	Single element character vector to convert

Details

By default R character vectors are converted to Python strings. In Python 3 these values are unicode objects however in Python 2 they are 8-bit string objects. This function enables you to obtain a Python unicode object from an R character vector when running under Python 2 (under Python 3 a standard Python string object is returned).

py_version *Python version*

Description

Get the version of Python currently being used by `reticulate`.

Usage

```
py_version()
```

Value

The version of Python currently used, or `NULL` if Python has not yet been initialized by `reticulate`.

r-py-conversion *Convert between Python and R objects*

Description

Convert between Python and R objects

Usage

```
r_to_py(x, convert = FALSE)

py_to_r(x)
```

Arguments

`x`	A Python object.
`convert`	Boolean; should Python objects be automatically converted to their R equivalent? If set to `FALSE`, you can still manually convert Python objects to R via the `py_to_r()` function.

Value

An R object, as converted from the Python object.

repl_python *Run a Python REPL*

Description

This function provides a Python REPL in the R session, which can be used to interactively run Python code. All code executed within the REPL is run within the Python main module, and any generated Python objects will persist in the Python session after the REPL is detached.

Usage

```
repl_python(
  module = NULL,
  quiet = getOption("reticulate.repl.quiet", default = FALSE),
  input = NULL
)
```

Arguments

module	An (optional) Python module to be imported before the REPL is launched.
quiet	Boolean; print a startup banner when launching the REPL? If TRUE, the banner will be suppressed.
input	Python code to be run within the REPL. Setting this can be useful if you'd like to drive the Python REPL programmatically.

Details

When working with R and Python scripts interactively, one can activate the Python REPL with `repl_python()`, run Python code, and later run `exit` to return to the R console.

Magics

A handful of magics are supported in `repl_python()`:

Lines prefixed with `!` are executed as system commands:

- `!cmd --arg1 --arg2`: Execute arbitrary system commands

Magics start with a `%` prefix. Supported magics include:

- `%conda ...` executes a conda command in the active conda environment
- `%pip ...` executes pip for the active python.
- `%load`, `%loadpy`, `%run` executes a python file.
- `%system`, `!!` executes a system command and capture output
- `%env`: read current environment variables.
  - `%env name`: read environment variable 'name'.
  - `%env name=val`, `%env name val`: set environment variable 'name' to 'val'. `val` elements in `{}` are interpolated using f-strings (required Python >= 3.6).

- `%cd <dir>` change working directory.
  - `%cd -`: change to previous working directory (as set by `%cd`).
  - `%cd -3`: change to 3rd most recent working directory (as set by `%cd`).
  - `%cd -foo/bar`: change to most recent working directory matching `"foo/bar"` regex (in history of directories set via `%cd`).
- `%pwd`: print current working directory.
- `%dhist`: print working directory history.

Additionally, the output of system commands can be captured in a variable, e.g.:

- `x = !ls`

where `x` will be a list of strings, consisting of stdout output split in `"\n"` (stderr is not captured).

Example

```
# enter the Python REPL, create a dictionary, and exit
repl_python()
dictionary = {'alpha': 1, 'beta': 2}
exit

# access the created dictionary from R
py$dictionary
# $alpha
# [1] 1
#
# $beta
# [1] 2
```

See Also

py, for accessing objects created using the Python REPL.

`source_python` *Read and evaluate a Python script*

Description

Evaluate a Python script within the Python main module, then make all public (non-module) objects within the main Python module available within the specified R environment.

Usage

```
source_python(file, envir = parent.frame(), convert = TRUE)
```

Arguments

file	The Python script to be executed.
envir	The environment to assign Python objects into (for example, parent.frame() or globalenv()). Specify NULL to not assign Python objects.
convert	Boolean; should Python objects be automatically converted to their R equivalent? If set to FALSE, you can still manually convert Python objects to R via the py_to_r() function.

Details

To prevent assignment of objects into R, pass NULL for the envir parameter.

---

tuple *Create Python tuple*

Description

Create a Python tuple object

Usage

```
tuple(..., convert = FALSE)
```

Arguments

...	Values for tuple (or a single list to be converted to a tuple).
convert	TRUE to automatically convert Python objects to their R equivalent. If you pass FALSE you can do manual conversion using the py_to_r() function.

Value

A Python tuple

Note

The returned tuple will not automatically convert its elements from Python to R. You can do manual conversion with the py_to_r() function or pass convert = TRUE to request automatic conversion.

use_python *Use Python*

Description

Select the version of Python to be used by reticulate.

Usage

```
use_python(python, required = NULL)

use_python_version(version, required = NULL)

use_virtualenv(virtualenv = NULL, required = NULL)

use_condaenv(condaenv = NULL, conda = "auto", required = NULL)

use_miniconda(condaenv = NULL, required = NULL)
```

Arguments

python	The path to a Python binary.
required	Is the requested copy of Python required? If TRUE, an error will be emitted if the requested copy of Python does not exist. If FALSE, the request is taken as a hint only, and scanning for other versions will still proceed. A value of NULL (the default), is equivalent to TRUE.
version	The version of Python to use. reticulate will search for versions of Python as installed by the install_python() helper function.
virtualenv	Either the name of, or the path to, a Python virtual environment.
condaenv	The conda environment to use. For use_condaenv(), this can be the name, the absolute prefix path, or the absolute path to the python binary. If the name is ambiguous, the first environment is used and a warning is issued. For use_miniconda(), the only conda installation searched is the one installed by install_miniconda().
conda	The path to a conda executable. By default, reticulate will check the PATH, as well as other standard locations for Anaconda installations.

Details

The reticulate package initializes its Python bindings lazily – that is, it does not initialize its Python bindings until an API that explicitly requires Python to be loaded is called. This allows users and package authors to request particular versions of Python by calling use_python() or one of the other helper functions documented in this help file.

RETICULATE_PYTHON

The `RETICULATE_PYTHON` environment variable can also be used to control which copy of Python `reticulate` chooses to bind to. It should be set to the path to a Python interpreter, and that interpreter can either be:

- A standalone system interpreter,
- Part of a virtual environment,
- Part of a Conda environment.

When set, this will override any other requests to use a particular copy of Python. Setting this in `~/.Renviron` (or optionally, a project `.Renviron`) can be a useful way of forcing `reticulate` to use a particular version of Python.

Caveats

Note that the requests for a particular version of Python via `use_python()` and friends only persist for the active session; they must be re-run in each new R session as appropriate.

If `use_python()` (or one of the other `use_*()` functions) are called multiple times, the most recently-requested version of Python will be used. Note that any request to `use_python()` will always be overridden by the `RETICULATE_PYTHON` environment variable, if set.

The `py_config()` function will also provide a short note describing why `reticulate` chose to select the version of Python that was ultimately activated.

`virtualenv-tools` *Interface to Python Virtual Environments*

Description

R functions for managing Python virtual environments.

Usage

```
virtualenv_create(
  envname = NULL,
  python = virtualenv_starter(version),
  ...,
  version = NULL,
  packages = "numpy",
  requirements = NULL,
  force = FALSE,
  module = getOption("reticulate.virtualenv.module"),
 system_site_packages = getOption("reticulate.virtualenv.system_site_packages", default
    = FALSE),
  pip_version = getOption("reticulate.virtualenv.pip_version", default = NULL),
 setuptools_version = getOption("reticulate.virtualenv.setuptools_version", default =
    NULL),
```

```
  extra = getOption("reticulate.virtualenv.extra", default = NULL)
)

virtualenv_install(
  envname = NULL,
  packages = NULL,
  ignore_installed = FALSE,
  pip_options = character(),
  requirements = NULL,
  ...,
  python_version = NULL
)

virtualenv_remove(envname = NULL, packages = NULL, confirm = interactive())

virtualenv_list()

virtualenv_root()

virtualenv_python(envname = NULL)

virtualenv_exists(envname = NULL)

virtualenv_starter(version = NULL, all = FALSE)
```

Arguments

envname	The name of, or path to, a Python virtual environment. If this name contains any slashes, the name will be interpreted as a path; if the name does not contain slashes, it will be treated as a virtual environment within `virtualenv_root()`. When `NULL`, the virtual environment as specified by the `RETICULATE_PYTHON_ENV` environment variable will be used instead. To refer to a virtual environment in the current working directory, you can prefix the path with `./<name>`.
python	The path to a Python interpreter, to be used with the created virtual environment. This can also accept a version constraint like `"3.10"`, which is passed on to `virtualenv_starter()` to find a suitable python binary.
...	Optional arguments; currently ignored and reserved for future expansion.
version, python_version	(string) The version of Python to use when creating a virtual environment. Python installations will be searched for using `virtualenv_starter()`. This can a specific version, like `"3.9"` or `"3.9.3"`, or a comma separated list of version constraints, like `">=3.8"`, or `"<=3.11,!=3.9.3,>3.6"`
packages	A set of Python packages to install (via `pip install`) into the virtual environment, after it has been created. By default, the `"numpy"` package will be installed, and the `pip`, `setuptools` and `wheel` packages will be updated. Set this to `FALSE` to avoid installing any packages after the virtual environment has been created.
requirements	Filepath to a pip requirements file.

`force` Boolean; force recreating the environment specified by `envname`, even if it already exists. If `TRUE`, the pre-existing environment is first deleted and then recreated. Otherwise, if `FALSE` (the default), the path to the existing environment is returned.

`module` The Python module to be used when creating the virtual environment – typically, `virtualenv` or `venv`. When `NULL` (the default), `venv` will be used if available with Python >= 3.6; otherwise, the `virtualenv` module will be used.

`system_site_packages` Boolean; create new virtual environments with the `--system-site-packages` flag, thereby allowing those virtual environments to access the system's site packages? Defaults to `FALSE`.

`pip_version` The version of `pip` to be installed in the virtual environment. Relevant only when `module == "virtualenv"`. Set this to `FALSE` to disable installation of `pip` altogether.

`setuptools_version` The version of `setuptools` to be installed in the virtual environment. Relevant only when `module == "virtualenv"`. Set this to `FALSE` to disable installation of `setuptools` altogether.

`extra` An optional set of extra command line arguments to be passed. Arguments should be quoted via `shQuote()` when necessary.

`ignore_installed` Boolean; ignore previously-installed versions of the requested packages? (This should normally be `TRUE`, so that pre-installed packages available in the site libraries are ignored and hence packages are installed into the virtual environment.)

`pip_options` An optional character vector of additional command line arguments to be passed to `pip`.

`confirm` Boolean; confirm before removing packages or virtual environments?

`all` If `TRUE`, `virtualenv_starter()` returns a 2-column data frame, with column names `path` and `version`. If `FALSE`, only a single path to a python binary is returned, corresponding to the first entry when `all = TRUE`, or `NULL` if no suitable python binaries were found.

Details

Virtual environments are by default located at `~/.virtualenvs` (accessed with the `virtualenv_root()` function). You can change the default location by defining the `WORKON_HOME` environment variable.

Virtual environments are created from another "starter" or "seed" Python already installed on the system. Suitable Pythons installed on the system are found by `virtualenv_starter()`.

`with.python.builtin.object` *Evaluate an expression within a context.*

Description

The `with` method for objects of type `python.builtin.object` implements the context manager protocol used by the Python `with` statement. The passed object must implement the context manager (`__enter__` and `__exit__` methods.

Usage

```
## S3 method for class 'python.builtin.object'
with(data, expr, as = NULL, ...)
```

Arguments

`data`	Context to enter and exit
`expr`	Expression to evaluate within the context
`as`	Name of variable to assign context to for the duration of the expression's evaluation (optional).
`...`	Unused

Index