

# Package ‘reticulate’

May 3, 2021

**Type** Package

**Title** Interface to 'Python'

**Version** 1.20

**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'  $\geq 2.7$ .

**License** Apache License 2.0

**URL** <https://github.com/rstudio/reticulate>

**BugReports** <https://github.com/rstudio/reticulate/issues>

**SystemRequirements** Python ( $\geq 2.7.0$ )

**Encoding** UTF-8

**Depends** R ( $\geq 3.0$ )

**Imports** Matrix, Rcpp ( $\geq 0.12.7$ ), graphics, jsonlite, methods, png, rappdirs, utils, withr

**Suggests** callr, knitr, rmarkdown, testthat

**LinkingTo** Rcpp

**RoxygenNote** 7.1.1

**VignetteBuilder** knitr

**NeedsCompilation** yes

**Author** Kevin Ushey [aut, cre],  
JJ Allaire [aut],  
RStudio [cph, fnd],  
Yuan Tang [aut, cph] (<<https://orcid.org/0000-0001-5243-233X>>),  
Dirk Eddelbuettel [ctb, cph],  
Bryan Lewis [ctb, cph],  
Sigrid Keydana [ctb],  
Ryan Hafen [ctb, cph],  
Marcus Geelnard [ctb, cph] (TinyThread library,  
<http://tinythreadpp.bitsnbites.eu/>)

**Maintainer** Kevin Ushey <kevin@rstudio.com>

**Repository** CRAN

**Date/Publication** 2021-05-03 04:50:05 UTC

## R topics documented:

array_reshape . . . . .	3
as.character.python.builtin.bytes . . . . .	4
configure_environment . . . . .	5
dict . . . . .	6
eng_python . . . . .	6
import . . . . .	7
install_miniconda . . . . .	9
install_python . . . . .	9
iterate . . . . .	10
miniconda_path . . . . .	11
miniconda_update . . . . .	11
np_array . . . . .	12
py . . . . .	12
PyClass . . . . .	13
py_available . . . . .	13
py_capture_output . . . . .	14
py_config . . . . .	15
py_del_attr . . . . .	15
py_del_item . . . . .	16
py_discover_config . . . . .	16
py_ellipsis . . . . .	17
py_eval . . . . .	17
py_exe . . . . .	18
py_func . . . . .	18
py_function_custom_scaffold . . . . .	19
py_get_attr . . . . .	20
py_get_item . . . . .	21
py_has_attr . . . . .	22
py_help . . . . .	22
py_id . . . . .	23
py_install . . . . .	23
py_is_null_xptr . . . . .	24
py_iterator . . . . .	25
py_last_error . . . . .	26
py_len . . . . .	27
py_list_attributes . . . . .	27
py_main_thread_func . . . . .	28
py_module_available . . . . .	28
py_none . . . . .	29
py_run . . . . .	29
py_save_object . . . . .	30

py_set_attr . . . . .	30
py_set_item . . . . .	31
py_set_seed . . . . .	31
py_str . . . . .	32
py_suppress_warnings . . . . .	32
py_unicode . . . . .	33
py_version . . . . .	33
r-py-conversion . . . . .	34
repl_python . . . . .	34
reticulate . . . . .	35
source_python . . . . .	36
tuple . . . . .	36
use_python . . . . .	37
virtualenv-tools . . . . .	38
with.python.builtin.object . . . . .	40
<b>Index</b>	<b>41</b>

---

array_reshape	<i>Reshape an Array</i>
---------------	-------------------------

---

## 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.

---

`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

<code>package</code>	The name of a package to configure. When <code>NULL</code> , <code>reticulate</code> will instead look at all loaded packages and discover their associated Python requirements.
<code>force</code>	Boolean; force configuration of the Python environment? Note that <code>configure_environment()</code> 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 `~/.Renv` 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 <code>py_to_r()</code> 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.

---

import	<i>Import a Python module</i>
--------	-------------------------------

---

**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)
```

```
import_builtins(convert = TRUE)
```

```
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 <code>py_to_r()</code> function.
delay_load	Boolean; delay loading the module until it is first used? When FALSE, the module will be loaded immediately. See <b>Delay Load</b> 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 list of named functions, which act as callbacks to be run when the module is later loaded. 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) { ... }
)
```

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 if 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	<i>Install Miniconda</i>
-------------------	--------------------------

---

**Description**

Download the [Miniconda](#) installer, and use it to install Miniconda.

**Usage**

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

**Arguments**

path	The path in which Miniconda will be installed. Note that the installer does not support paths containing spaces. See <a href="#">miniconda_path</a> for more details on the default path used by reticulate.
update	Boolean; update to the latest version of Miniconda after install?
force	Boolean; force re-installation if Miniconda is already installed at the requested path?

**See Also**

Other miniconda: [miniconda\\_path\(\)](#), [miniconda\\_update\(\)](#)

---

install_python	<i>Install Python</i>
----------------	-----------------------

---

**Description**

Download and install Python, using the [pyenv](#). and [pyenv-win](#) projects.

**Usage**

```
install_python(version, list = FALSE, force = FALSE)
```

**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?

**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 <- "3.8.7"
install_python(version = version)
virtualenv_create("my-environment", python_version = version)
use_virtualenv("my-environment", required = TRUE)
```

---

 iterate

*Traverse a Python iterator or generator*


---

**Description**

Traverse a Python iterator or generator

**Usage**

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

```
iter_next(it, completed = NULL)
```

```
as_iterator(x)
```

**Arguments**

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

**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).

---

miniconda_path	<i>Path to Miniconda</i>
----------------	--------------------------

---

**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()
```

**See Also**

Other miniconda: [install\\_miniconda\(\)](#), [miniconda\\_update\(\)](#)

---

miniconda_update	<i>Update Miniconda</i>
------------------	-------------------------

---

**Description**

Update Miniconda to the latest version.

**Usage**

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

**Arguments**

path	The path in which Miniconda will be installed. Note that the installer does not support paths containing spaces.
------	--

**See Also**

Other miniconda: [install\\_miniconda\(\)](#), [miniconda\\_path\(\)](#)

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	<i>Create a python class</i>
---------	------------------------------

---

**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 <code>python.builtin.type</code> 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	<i>Check if Python is available on this system</i>
--------------	--

---

**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_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_config	<i>Python configuration</i>
-----------	-----------------------------

---

**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	<i>Delete an attribute of a Python object</i>
-------------	---

---

**Description**

Delete an attribute of a Python object

**Usage**

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

**Arguments**

x	A Python object.
name	The attribute name.

---

py\_del\_item                      *Delete / remove an item from a Python object*

---

**Description**

Delete an item associated with a Python object, as through its `__delitem__` method.

**Usage**

```
py_del_item(x, name)
```

**Arguments**

x	A Python object.
name	The item name.

**Value**

The (mutated) object x, invisibly.

**See Also**

Other item-related APIs: [py\\_get\\_item\(\)](#), [py\\_set\\_item\(\)](#)

---

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 must be available in order for a version of Python to be used.
use_environment	An optional virtual/conda environment name to prefer in the search

**Value**

Python configuration object.

---

py_ellipsis	<i>The builtin constant Ellipsis</i>
-------------	--------------------------------------

---

**Description**

The builtin constant Ellipsis

**Usage**

```
py_ellipsis()
```

---

py_eval	<i>Evaluate a Python Expression</i>
---------	-------------------------------------

---

**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 associated with the instance currently being used by `reticulate`.

### 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 associated Python executable, or `NULL` if Python has not yet been initialized.

---

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_function` before executing `python_function`.

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	<i>Get an item from a Python object</i>
-------------	---

---

**Description**

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

**Usage**

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

**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.

**See Also**

Other item-related APIs: [py\\_del\\_item\(\)](#), [py\\_set\\_item\(\)](#)

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**

<code>x</code>	A python object.
<code>name</code>	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**

<code>object</code>	Object to print documentation for
---------------------	-----------------------------------

---

py_id	<i>Unique identifier for Python object</i>
-------	--

---

**Description**

Get a globally unique identifier for a Python object.

**Usage**

```
py_id(object)
```

**Arguments**

object	Python object
--------	---------------

**Value**

Unique identifier (as integer) or NULL

**Note**

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

---

py_install	<i>Install Python packages</i>
------------	--------------------------------

---

**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,  
    ...  
)
```

**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 <code>r-reticulate</code> 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 <b>Finding Conda</b> 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 <code>conda_install()</code> or <code>virtualenv_install()</code> .

**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-tools](#), [virtualenv-tools](#)

---

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

---

<code>py_iterator</code>	<i>Create a Python iterator from an R function</i>
--------------------------	--

---

**Description**

Create a Python iterator from an R function

**Usage**

```
py_iterator(fn, completed = NULL)
```

**Arguments**

<code>fn</code>	R function with no arguments.
<code>completed</code>	Special sentinel return value which indicates that iteration is complete (defaults to <code>NULL</code> )

**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 parallelize 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\_last\_error

*Get or clear the last Python error encountered*

---

**Description**

Get or clear the last Python error encountered

**Usage**

```
py_last_error()
```

```
py_clear_last_error()
```

**Value**

For `py_last_error()`, a list with the type, value, and traceback for the last Python error encountered (can be NULL if no error has yet been encountered).

---

py_len	<i>Length of Python object</i>
--------	--------------------------------

---

**Description**

Get the length of a Python object (equivalent to the Python len() built in function).

**Usage**

```
py_len(x)
```

**Arguments**

x	Python object
---	---------------

**Value**

Length as integer

---

py_list_attributes	<i>List all attributes of a Python object</i>
--------------------	---

---

**Description**

List all attributes of a Python object

**Usage**

```
py_list_attributes(x)
```

**Arguments**

x	Python object
---	---------------

**Value**

Character vector of attributes

---

`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**

Check if a Python module is available on this system.

**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	<i>The Python None object</i>
---------	-------------------------------

---

**Description**

Get a reference to the Python None object.

**Usage**

```
py_none()
```

---

py_run	<i>Run Python code</i>
--------	------------------------

---

**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)
```

**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 <a href="#">py_to_r()</a> function.
file	The Python script to be executed.

**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 with pickle*

---

**Description**

Save and load Python objects with pickle

**Usage**

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

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

**Arguments**

object	Object to save
filename	File name
pickle	The implementation of pickle to use (defaults to "pickle" but could e.g. also be "cPickle")
...	Optional arguments to be passed to the load() function defined by the associated pickle module.

---

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_item	<i>Set an item for a Python object</i>
-------------	--

---

**Description**

Set an item on a Python object, similar to how `x[name] = value` might be used in Python code to set an item called `name` with value `value` on object `x`. The object's `__setitem__` method will be called.

**Usage**

```
py_set_item(x, name, value)
```

**Arguments**

<code>x</code>	A Python object.
<code>name</code>	The item name.
<code>value</code>	The item value.

**Value**

The (mutated) object `x`, invisibly.

**See Also**

Other item-related APIs: [py\\_del\\_item\(\)](#), [py\\_get\\_item\(\)](#)

---

py_set_seed	<i>Set Python and NumPy random seeds</i>
-------------	--

---

**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**

<code>seed</code>	A single value, interpreted as an integer
<code>disable_hash_randomization</code>	Disable hash randomization, which is another common source of variable results. See <a href="https://docs.python.org/3.4/using/cmdline.html#envvar-PYTHONHASHSEED">https://docs.python.org/3.4/using/cmdline.html#envvar-PYTHONHASHSEED</a>

**Details**

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

---

py_str	<i>An S3 method for getting the string representation of a Python object</i>
--------	--

---

**Description**

An S3 method for getting the string representation of a Python object

**Usage**

```
py_str(object, ...)
```

**Arguments**

object	Python object
...	Unused

**Details**

The default implementation will call PyObject\_Str on the object.

**Value**

Character vector

---

py_suppress_warnings	<i>Suppress Python warnings for an expression</i>
----------------------	---

---

**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	<i>Convert to Python Unicode Object</i>
------------	---

---

**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	<i>Python version</i>
------------	-----------------------

---

**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	<i>Convert between Python and R objects</i>
-----------------	---

---

**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 <a href="#">py_to_r()</a> function.

**Value**

An R object, as converted from the Python object.

---

repl_python	<i>Run a Python REPL</i>
-------------	--------------------------

---

**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.

## See Also

[py](#), for accessing objects created using the Python REPL.

## Examples

```
## Not run:

# 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

## End(Not run)
```

## Description

R 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. The reticulate package is compatible with all versions of Python  $\geq 2.7$ . Integration with NumPy requires NumPy version 1.6 or higher.

---

source_python	<i>Read and evaluate a Python script</i>
---------------	--

---

**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, <code>parent.frame()</code> or <code>globalenv()</code> ). Specify <code>NULL</code> to not assign Python objects.
convert	Boolean; should Python objects be automatically converted to their R equivalent? If set to <code>FALSE</code> , you can still manually convert Python objects to R via the <a href="#">py_to_r()</a> function.

**Details**

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

---

tuple	<i>Create Python tuple</i>
-------	----------------------------

---

**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 <code>FALSE</code> you can do manual conversion using the <a href="#">py_to_r()</a> 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 = FALSE)
```

```
use_python_version(version, required = FALSE)
```

```
use_virtualenv(virtualenv = NULL, required = FALSE)
```

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

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

**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. Otherwise, the request is taken as a hint only, and scanning for other versions will still proceed.
version	The version of Python to use. reticulate will search for versions of Python as installed by the <code>install_python()</code> helper function.
virtualenv	Either the name of, or the path to, a Python virtual environment.
condaenv	The name of the Conda environment to use.
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 `~/.Renviro` (or optionally, a `project .Renviro`) can be a useful way of forcing reticulate to use a particular version of Python.

## Caveats

By default, requests are *advisory*, and may be ignored for a number of reasons:

- The requested copy of Python cannot be initialized,
- The requested copy of Python does not have an installation of numpy available,
- Another call to `use_python()` has requested a different version of Python,
- The request has been overridden via `use_python(..., required = TRUE)`.

In general, if you explicitly want to use a particular version of Python, it is recommended to set `required = TRUE`, or explicitly set the RETICULATE\_PYTHON environment variable.

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.

---

virtualenv-tools

*Interface to Python Virtual Environments*

---

## Description

R functions for managing Python **virtual environments**.

## Usage

```
virtualenv_create(
  envname = NULL,
  python = NULL,
  ...,
  version = NULL,
  packages = "numpy",
  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("reticulat.evirtualenv.extra", default = NULL)
)

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

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

virtualenv_list()

virtualenv_root()

virtualenv_python(envname = NULL)

virtualenv_exists(envname = NULL)

```

## 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 <code>virtualenv_root()</code> . When NULL, the virtual environment as specified by the <code>RETICULATE_PYTHON_ENV</code> environment variable will be used instead. To refer to a virtual environment in the current working directory, you can prefix the path with <code>./&lt;name&gt;</code> .
python	The path to a Python interpreter, to be used with the created virtual environment. When NULL, the Python interpreter associated with the current session will be used.
...	Optional arguments; currently ignored and reserved for future expansion.
version	The version of Python to be used with the newly-created virtual environment. Python installations as installed via <code>install_python()</code> will be used.
packages	A set of Python packages to install (via <code>pip install</code> ) into the virtual environment, after it has been created. By default, the "numpy" package will be installed, and the <code>pip</code> , <code>setuptools</code> and <code>wheel</code> packages will be updated. Set this to <code>FALSE</code> to avoid installing any packages after the virtual environment has been created.
module	The Python module to be used when creating the virtual environment – typically, <code>virtualenv</code> or <code>venv</code> . When NULL (the default), <code>venv</code> will be used if available with Python <code>&gt;= 3.6</code> ; otherwise, the <code>virtualenv</code> module will be used.
system_site_packages	Boolean; create new virtual environments with the <code>--system-site-packages</code> flag, thereby allowing those virtual environments to access the system's site packages? Defaults to <code>FALSE</code> .

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?

### 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.

---

```
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

- \* **datasets**
  - py, [12](#)
- \* **item-related APIs**
  - py\_del\_item, [16](#)
  - py\_get\_item, [21](#)
  - py\_set\_item, [31](#)
- \* **miniconda**
  - install\_miniconda, [9](#)
  - miniconda\_path, [11](#)
  - miniconda\_update, [11](#)
- array\_reshape, [3](#)
- as.character.python.builtin.bytes, [4](#)
- as\_iterator(iterate), [10](#)
- conda-tools, [24](#)
- conda\_install(), [24](#)
- configure\_environment, [5](#)
- dict, [6](#)
- eng\_python, [6](#)
- import, [7](#)
- import\_builtins(import), [7](#)
- import\_from\_path(import), [7](#)
- import\_main(import), [7](#)
- install\_miniconda, [9](#), [11](#)
- install\_python, [9](#)
- install\_python(), [10](#), [37](#), [39](#)
- iter\_next(iterate), [10](#)
- iterate, [10](#)
- miniconda\_path, [9](#), [11](#), [11](#)
- miniconda\_update, [9](#), [11](#), [11](#)
- np\_array, [12](#)
- py, [12](#), [35](#)
- py\_available, [13](#)
- py\_capture\_output, [14](#)
- py\_clear\_last\_error(py\_last\_error), [26](#)
- py\_config, [15](#)
- py\_del\_attr, [15](#)
- py\_del\_item, [16](#), [21](#), [31](#)
- py\_dict(dict), [6](#)
- py\_discover\_config, [16](#)
- py\_discover\_config(), [15](#)
- py\_ellipsis, [17](#)
- py\_eval, [17](#)
- py\_exe, [18](#)
- py\_func, [18](#)
- py\_function\_custom\_scaffold, [19](#)
- py\_function\_docs(), [19](#)
- py\_get\_attr, [20](#)
- py\_get\_attr(), [8](#)
- py\_get\_item, [16](#), [21](#), [31](#)
- py\_has\_attr, [22](#)
- py\_help, [22](#)
- py\_id, [23](#)
- py\_install, [23](#)
- py\_is\_null\_xptr, [24](#)
- py\_iterator, [25](#)
- py\_last\_error, [26](#)
- py\_len, [27](#)
- py\_list\_attributes, [27](#)
- py\_load\_object(py\_save\_object), [30](#)
- py\_main\_thread\_func, [28](#)
- py\_module\_available, [28](#)
- py\_none, [29](#)
- py\_numpy\_available(py\_available), [13](#)
- py\_run, [29](#)
- py\_run\_file(py\_run), [29](#)
- py\_run\_string(py\_run), [29](#)
- py\_run\_string(), [18](#)
- py\_save\_object, [30](#)
- py\_set\_attr, [30](#)
- py\_set\_item, [16](#), [21](#), [31](#)
- py\_set\_seed, [31](#)
- py\_str, [32](#)

- py\_suppress\_warnings, 32
- py\_to\_r (r-py-conversion), 34
- py\_to\_r(), 6, 7, 29, 34, 36, 37
- py\_unicode, 33
- py\_validate\_xptr (py\_is\_null\_xptr), 24
- py\_version, 33
- PyClass, 13
  
- r-py-conversion, 34
- r\_to\_py (r-py-conversion), 34
- repl\_python, 34
- reticulate, 35
  
- set.seed(), 32
- source\_python, 36
  
- tuple, 36
  
- use\_condaenv (use\_python), 37
- use\_miniconda (use\_python), 37
- use\_python, 37
- use\_python\_version (use\_python), 37
- use\_virtualenv (use\_python), 37
  
- virtualenv-tools, 24, 38
- virtualenv\_create (virtualenv-tools), 38
- virtualenv\_exists (virtualenv-tools), 38
- virtualenv\_install (virtualenv-tools),  
38
- virtualenv\_install(), 24
- virtualenv\_list (virtualenv-tools), 38
- virtualenv\_python (virtualenv-tools), 38
- virtualenv\_remove (virtualenv-tools), 38
- virtualenv\_root (virtualenv-tools), 38
  
- with.python.builtin.object, 40