Package 'PythonInR'

April 23, 2019

Title Use 'Python' from Within 'R'

Version 0.1-7

Description Interact with 'Python' <https://www.python.org/> from within 'R'.

Depends R (>= 2.15)

Imports pack, methods, R6, stats

SystemRequirements Python (>= 2.7.0) with header files and shared library (UNIX) / static library (Windows)

Note More information about the system requirements can be found in the README file.

URL https://bitbucket.org/Floooo/pythoninr/

BugReports https://bitbucket.org/Floooo/pythoninr/issues

License GPL-3

NeedsCompilation yes

RoxygenNote 6.0.0

Author Florian Schwendinger [aut, cre], Kimyen Truong [ctb]

Maintainer Florian Schwendinger <FlorianSchwendinger@gmx.at>

Repository CRAN

Date/Publication 2019-04-23 07:10:03 UTC

R topics documented:

`autodetectPython` *Autodetects the settings for Windows*

Description

Autodetects the settings needed to connect to the python dll file (**only Windows**).

Usage

```
autodetectPython(pythonExePath = NULL)
```

Arguments

`pythonExePath`	a string containing the path to "python.exe" (e.g. "C:\Python34\python.exe").

Value

Returns a list containing the information necessary to connect to Python if a compatible Python, version was found, raises an error otherwise.

Examples

```
## Not run:
  autodetectPython()
  autodetectPython("C:\\Python27\\python.exe")

## End(Not run)
```

BEGIN.Python *Execute Python interactively from within R*

Description

The function BEGIN.Python starts an Python read-eval-print loop.

Usage

```
BEGIN.Python()
```

Details

BEGIN.Python emulates the behavior of the Python terminal and therefore allows interactive Python code development from within R.

Value

Returns the entered code as character, code lines which throw an exception are omitted.

Note

This won't work with RStudio because of a known RStudio issue.

Examples

```
## Not run:
code <-
BEGIN.Python()
import os
os.getcwd()
dir(os)
x = 3**3
for i in xrange(10):
    if (i > 5):
        print(i)

END.Python
## NOTE: BEGIN.Python returns the successfully executed code as character.
cat(code, sep="\n")
pyGet0("x")

## End(Not run)
```

pyAttach *Attach Python objects to R*

Description

A convenience function to attach Python objects to R.

Usage

```
pyAttach(what, env = parent.frame())
```

Arguments

what	a character vector giving the names of the Python objects, which should be attached to R.
env	the environment where the virtual Python objects are assigned to.

Examples

```
if ( pyIsConnected() ){
pyExec("import os")

## attach to global
## ----------------
## attach the function getcwd from the module os to R.
pyAttach("os.getcwd", .GlobalEnv)
os.getcwd
os.getcwd()
## attach the string object os.name to R
pyAttach("os.name", .GlobalEnv)
pyExecp("os.name")
os.name
## Since os.name is attached to the globalenv it can be set without using
## the global assignment operator
os.name = "Hello Python from R!"
pyExecp("os.name")
os.name
## Please note if you don't pyAttach to globalenv you have to use
## the global assignment operator to set the values of the Python objects

## attach to a new environment
## ---------------------------
os <- new.env()
attach(os, name="python:os")
pyAttach(paste("os", pyDir("os"), sep="."), as.environment("python:os"))
os.sep
os.sep = "new sep" ## this doesn't changes the value in Python but only
                   ## assigns the new variable os.sep to globalenv
```

```
os.sep
.GlobalEnv$`os.sep`
as.environment("python:os")$`os.sep`
pyExecp("os.sep")
ls()
ls("python:os")
os.sep <<- "this changes the value in Python"
.GlobalEnv$`os.sep`
as.environment("python:os")$`os.sep`
pyExecp("os.sep")
}
```

---

pyCall *Call a callable Python object from within R*

Description

Call a callable Python object from within R.

Usage

```
pyCall(callableObj, args = NULL, kwargs = NULL, autoTypecast = TRUE,
  simplify = TRUE)
```

Arguments

callableObj	a character string giving the name of the desired callable Python object.
args	an optional list of arguments passed to the callable.
kwargs	an optional list of named arguments passed to the callable.
autoTypecast	an optional logical value, default is TRUE, specifying if the return values should be automatically typecasted if possible.
simplify	an optional logical value, if TRUE, R converts Python lists into R vectors whenever possible, else it translates Python lists always to R lists.

Details

The args and kwargs are transformed to Python variables by the default conversion. More information about the type conversion can be found in the vignette.

Value

Returns the result of the function call, converted into an R object.

Examples

```
pyCall("sum", args=list(1:3))

## define a new function with the name fun
pyExec('
def fun(**kwargs):
    return([(key, value) for key, value in kwargs.items()])
')
pyCall("fun", kwargs=list(a=1, f=2, x=4))
```

---

pyConnect *connects R to Python*

Description

Connects R to Python. **(The parameters are only needed for the Windows version!)**

Usage

```
pyConnect(pythonExePath = NULL, dllDir = NULL, pythonHome = NULL)

pyConnectWinDll(dllName, dllDir, majorVersion, pythonHome, pyArch,
  useCstdout = NULL)
```

Arguments

pythonExePath	a character containing the path to "python.exe" (e.g. "C:\Python27\python.exe")
dllDir	an optional character giving the path to the dll file. Since the dll file is normally in a system folder or in the same location as python.exe, this parameter is **almost never needed**!
pythonHome	an optional character giving the path to PYTHONHOME. On Windows by default PYTHONHOME is the folder where python.exe is located, therefore this parameter is **normally not needed**.
dllName	a character giving the name of the dll file (e.g.d "python27.dll").
majorVersion	an integer giving the major Python version (e.g. 2 or 3).
pyArch	a character giving the Python architecture, i.e. "32bit" or "64bit".
useCstdout	a logical indicating if the C stdout should be used or the stout should be redirected at a Python level.

Details

There is a different behavior for the static (Linux default) and the explicit linked (Windows default) version. Where as the static linked version automatically connects, when the package get's loaded, the explicitly linked version needs to be connected manually. More information can be found at the README file or at http://pythoninr.bitbucket.org/.

Note

See the **README** for more information about the Windows setup.

Examples

```
## Not run:
## Linux examples
pyConnect() # is done by default when the package is loaded

## Windows examples
pyConnect() ## will try to detect a suitable python version
            ## from the PATH given in the environment variables
pyConnect("C:\\Python27\\python.exe")

## One can also explicitly set the parameters for the connection.
PythonInR:::pyConnectWinDll(dllName="python27.dll", dllDir=NULL,
                            majorVersion=2, pythonHome="C:\\Python27",
                            pyArch="32bit")

## End(Not run)
```

---

pyDict *Create a virtual Python dictionary*

Description

The function pyDict creates a virtual Python object of type PythonInR_Dict.

Usage

```
pyDict(key, value, regFinalizer = TRUE)
```

Arguments

key	a character string giving the name of the Python object.
value	if a value is provided, a new Python dictionary is created based on the value. Therefore allowed values of value are named lists and names vectors.
regFinalizer	a logical indicating if a finalizer should be be registered, the default value is TRUE.

Details

If no value is provided a virtual Python dict for an existing Python object is created. If the value is NULL, an empty virtual Python object for an empty dict is created. If the value is a named vector or named list, a new Python object based on the vector or list is created.

Examples

```
if ( pyIsConnected() ){
pyExec('myPyDict = {"a":1, "b":2, "c":3}')
## create a virtual Python dictionary for an existing dictionary
myDict <- pyDict("myPyDict")
myDict["a"]
myDict["a"] <- "set the key"
myDict
## allowed keys are
myDict['string'] <- 1
myDict[3L] <- "long"
myDict[5] <- "float"
myDict[c("t", "u", "p", "l", "e")] <- "tuple"
myDict
## NOTE: Python does not make a difference between a float key 3 and a long key 3L
myDict[3] <- "float"
myDict
## create a new Python dict and virtual dict
myNewDict <- pyDict('myNewDict', list(p=2, y=9, r=1))
myNewDict
}
```

---

pyDir *Convenience function to call the Python function* **dir**

Description

A convenience function to call the Python function **dir**.

Usage

```
pyDir(objName = NULL)
```

Arguments

objName	an optional string specifying the name of the Python object.

Details

The Python function dir is similar to the R function ls.

Value

Returns the list of names in the global scope, if no object name is provided, otherwise a list of valid attributes for the specified object.

Examples

```
pyDir()
pyDir("sys")
```

---

pyExec *Executes multiple lines of Python code from within R*

Description

The function pyExec allows to execute multiple lines of python code from within R.

Usage

```
pyExec(code)
```

Arguments

code	a string of Python code to be executed in Python.

Details

Since pyExec can execute multiple lines, it is the obvious choice for defining Python functions or running small scripts where no return value is needed.

Examples

```
pyExec('
print("The following line will not appear in the R terminal!")
"Hello" + " " + "R!"
print("NOTE: pyExecp would also show the line above!")
print("The following line will appear in the R terminal!")
print("Hello" + " " + "R!")
')
```

---

pyExecfile *Executes Python source file from within R*

Description

The function pyExecfile calls the Python function `execfile`. which is the Python equivalent to the function `source` provided in R.

Usage

```
pyExecfile(filename)
```

Arguments

filename	a character string giving the name or full path of the file to be executed.

Details

The function execfile is kind of the source of Python. Since it got omitted in Python 3 a replacement gets assigned following common practices.

Examples

```
## Not run:
pyExecfile("myPythonScript.py")

## End(Not run)
```

---

pyExecg *Executes multiple lines of python code and gets the output*

Description

The function pyExecg is designed to execute multiple lines of Python code and returns the thereby generated variables to R.

Usage

```
pyExecg(code, returnValues = character(), autoTypecast = TRUE,
  returnToR = TRUE, mergeNamespaces = FALSE, override = FALSE,
  simplify = TRUE)
```

Arguments

code	a string of Python code to be executed in Python.
returnValues	a character vector containing the names of the variables, which should be returned to R.
autoTypecast	a an optional logical value, default is TRUE, specifying if the return values should be automatically typecasted if possible.
returnToR	an optional logical, default is TRUE, specifying if the generated variables should be returned to R.
mergeNamespaces	an optional logical, default is FALSE, specifying if the internally generated temporary namespace should be merged with the name space __main__. See **Details**.
override	an optional logical value, default is FALSE, specifying how to merge the temporary namespace with the __main__ namespace.
simplify	an optional logical, if TRUE (default) R converts Python lists into R vectors whenever possible, else it translates Python lists always to R lists.

Details

The function pyExecg executes the code in a temporary namespace, after the execution every variable from the namespace is returned to R. If the mergeNamespaces is set to TRUE the temporary namespace gets merged with the (global) namespace __main__. The logical variable override is used to control, if already existing variables in the namespace __main__ should be overridden, when a variable with the same name get's assigned to the temporary namespace. If a python object can't be converted to an R object it is assigned to the Python dictionary __R__.namespace and the type, id and an indicator if the object is a callable are returned.

Value

Returns a list containing all the variables of the __main__ namespace.

Examples

```
if ( pyIsConnected() ){
# 1. assigns x to the global namespace
pyExec("x=4")
# 2. assigns y to the temp namespace
pyExecg("y=4", simplify=TRUE)
# 3. assign again to the temp namespace
pyExecg("
y=[i for i in range(1,4)]
x=[i for i in range(3,9)]
z=[i**2 for i in range(1,9)]
", returnValues=c("x", "z"), simplify=TRUE)
# 4. assign x to the temp namespace, x gets returned as vector
pyExecg("x=[i for i in range(0,5)]", simplify=TRUE)
# 5. assign x to the temp namespace, x gets returned as list
pyExecg("x=[i for i in range(0,5)]", simplify=FALSE)
# 6. x is still 4 since except assignment 1 all other assignments
#    took place in the temp namespace
pyPrint("x")
# 7. note y has never been assigned to the main namespace
"y" %in% pyDir()
# 8. since mergeNamespaces is TRUE PythonInR will try
#     to assign x to the main namespace but since override is
#     by default FALSE and x already exists in the main namespace
#     x will not be changed
pyExecg("x=10", simplify=TRUE, mergeNamespaces=TRUE)
# 9. there is no y in the main namespace therefore it can be assigned
pyExecg("y=10", simplify=TRUE, mergeNamespaces=TRUE)
pyPrint("x") # NOTE: x is still unchanged!
pyPrint("y") # NOTE: a value has been assigned to y!
# 10. since override is now TRUE the value of x will be changed in the
#      main namespace
pyExecg("x=10", simplify=TRUE, mergeNamespaces=TRUE, override=TRUE)
pyPrint("x") # NOTE: x is changed now!
# 11. get an object which can't be typecast to an R object
#     pyExecg does not transform these objects automatically
```

```
pyExec("import os")
z <- pyExecg("x = os")
os <- PythonInR:::pyTransformReturn(z[[1]])
os$getcwd()
}
```

---

pyExecp *Executes a single line of Python code from within R*

Description

The function pyExecp is designed to execute a single line of Python code from within R. Thereby pyExecp tries to emulate the natural interactive Python terminal behavior.

Usage

```
pyExecp(code)
```

Arguments

code	a string of Python code to be executed in Python.

Details

The name pyExecp is short for python execute and print. As the name is indicating the most visual difference between pyExec and pyExecp lies in the printing behavior. For example, executing `pyExecp('"Hello " + "R!"')` would show `'Hello R!'` in the R terminal, executing `pyExec('"Hello " + "R!"')` wouldn't show anything. Internally pyExec uses PyRun_SimpleString and pyExecp uses PyRun_String with the flag Py_single_input, therefore pyExecp can be used to simulate an interactive Python interpreter behavior.

Examples

```
pyExecp('"Hello" + " " + "R!"')
```

---

pyExit *closes the connection to Python*

Description

Closes the connection from R to Python.

Usage

```
pyExit()
```

Examples

```
## Not run:
pyExit()

## End(Not run)
```

---

pyFunction *creates a virtual Python function*

Description

The function pyFunction creates a new object of type pyFunction based on a given key.

Usage

```
pyFunction(key, regFinalizer = FALSE)
```

Arguments

key	a string specifying the name of a Python method/function.
regFinalizer	a logical indicating if a finalizer should be be registered, the default value is FALSE.

Details

The function pyFunction makes it easy to create interfaces to Python functions.

Examples

```
if ( pyIsConnected() ){
pySum <- pyFunction("sum")
pySum(1:3)
}
```

---

pyGet *Gets Python objects by name and transforms them into R objects*

Description

The function pyGet gets Python objects by name and transforms them into R objects.

Usage

```
pyGet(key, autoTypecast = TRUE, simplify = TRUE)
```

Arguments

key	a string specifying the name of a Python object.
autoTypecast	an optional logical value, default is TRUE, specifying if the return values should be automatically typecasted if possible.
simplify	an optional logical value, if TRUE R converts Python lists into R vectors whenever possible, else it translates Python lists always into R lists.

Details

Since any Python object can be transformed into one of the basic data types it is up to the user to do so up front. More information about the type conversion can be found in the README file or at http://pythoninr.bitbucket.org/.

Value

Returns the specified Python object converted into an R object if possible, else a virtual Python object.

Note

pyGet always returns a new object, if you want to create a R representation of an existing Python object use pyGet0 instead.

Examples

```
## get a character of length 1
pyGet("__name__")
## get a character of length 1 > 1
pyGet("sys.path")
## get a list
pyGet("sys.path", simplify = FALSE)
## get a PythonInR_List
x <- pyGet("sys.path", autoTypecast = FALSE)
x
class(x)

## get an object where no specific transformation to R is defined
## this example also shows the differnces between pyGet and pyGet0
pyExec("import datetime")
## pyGet creates a new Python variable where the return value of pyGet is
## stored the name of the new reference is stored in x$py.variableName.
x <- pyGet("datetime.datetime.now().time()")
x
class(x)
x$py.variableName
## pyGet0 never creates a new Python object, objects which can be transformed
## to R objects are transformed. For all other objects an PythonInR_Object is created.
y <- pyGet0("datetime.datetime.now().time()")
```

```
y
## An important difference is that the evaluation of x always will return the same
## time, the evaluation of y always will give the new time.
```

---

pyGet0 *Creates an R representation of an Python object*

Description

The function pyGet0 gets Python objects by name.

Usage

```
pyGet0(key)
```

Arguments

key	a string specifying the name of a Python object.

Details

Primitive data types like bool, int, long, float, str, bytes and unicode are returned as R objects. Python tuples, lists, dictionaries and other Python objects are returned as virtual Python objects.

Value

Returns the specified Python object converted into an R object if possible, else a virtual Python object.

Note

pyGet0 never creates a new Python object.

Examples

```
if ( pyIsConnected() ){
pyExec("import os")
os <- pyGet0("os")
os$getcwd()
os$sep
os$sep <- "Hello Python!"
pyExecp("os.sep")
}
```

pyHelp *Convenience function to access the Python* **help** *system*

Description

a convenience function to access the Python **help** system.

Usage

```
pyHelp(topic)
```

Arguments

topic	a string specifying name or topic for which help is sought.

Value

Prints the help to the given string.

Examples

```
pyHelp("abs")
```

pyImport *Import virtual Python objects to R*

Description

A convenience function to call the Python function **import** and creating virtual Python objects for the imported objects in R.

Usage

```
pyImport(import, from = NULL, as = NULL, env = parent.frame())
```

Arguments

import	a character giving the names of the objects to import.
from	an optional character string giving the name of the module.
as	an optional string defining an alias for the module name.
env	an optional environment where the virtual Python objects are assigned to.

Details

The function pyImport works like the import function in Python.

The function pyImport has a special behavior for the packages numpy and pandas. For these two packages pyImport does not only import numpy but also register their alias in pyOptions. To be found when pySet is used with the option useNumpy set to TRUE.

Examples

```
pyImport("os")
## Not run:
#NOTE: The following does not only import numpy but also register the
#      alias in the options under the name "numpyAlias".
#      The same is done for pandas, the default alias for pandas and numpy
#      are respectively "pandas" and "numpy". The numpyAlias is used
#      when calling pySet with the pyOption useNumpy set to TRUE.
pyOptions("numpyAlias")
pyImport("numpy", as="np")
pyOptions("numpyAlias")
pyImport("pandas", as="pd")
pyImport(c("getcwd", "sep"), from="os")
getcwd()
sep
sep = "Hello R!"
pyExecp("sep")

## End(Not run)
```

---

pyIsConnected *checks if R is connected to Python*

Description

Checks if R is connected to Python.

Usage

```
pyIsConnected()
```

Value

Returns TRUE if R is connected to Python, FALSE otherwise.

Examples

```
pyIsConnected()
```

pyList *Creates a virtual Python list*

Description

The function pyList creates a virtual Python object of type PythonInR_List.

Usage

```
pyList(key, value, regFinalizer = TRUE)
```

Arguments

key	a character string giving the name of the Python object.
value	an optional value, allowed values are vectors, lists and NULL.
regFinalizer	a logical indicating if a finalizer should be be registered, the default value is TRUE.

Details

If no value is provided a virtual Python list for an existing Python object is created. If the value is NULL an empty virtual Python object for an empty list is created. If the value is a vector or a list, a new Python object based on the vector or list is created.

Examples

```
if ( pyIsConnected() ){
pyExec('myPyList = [1, 2, 5, "Hello R!"]')
# create a virtual Python list for an existing list
myList <- pyList("myPyList")
myList[0]
myList[1] <- "changed"
myList
# create a new Python list and virtual list
myNewList <- pyList('myNewList', list(1:3, 'Hello Python'))
myNewList[1]
myNewList$append(4L)
ls(myNewList)
## NOTE: Indexing which can not be interpreted as correct R
##       syntax should be provided as a character string.
myNewList['::2']
}
```

pyObject *Creates a virtual Python object*

Description

The function pyObject creates a virtual Python object of type PythonInR_Object.

Usage

```
pyObject(key, regFinalizer = TRUE)
```

Arguments

key	a character string giving the name of the Python object.
regFinalizer	a logical indicating if a finalizer should be be registered, the default value is TRUE.

Details

Every PythonInR_Object has the following members:

- **py.variableName** the variable name used in Python.
- **py.objectName** the name of the Python object (obtained by x.__name__) or NULL.
- **py.type** the type of the Python object.
- **py.del** a function to delete the Python object.
- **print** for more information see R6 classes.
- **initialize** for more information see R6 classes.

The other members of PythonInR_Object's are generated dynamically based on the provided Python object. The R function **ls** can be used to view the members of a PythonInR_Object object.

Examples

```
if ( pyIsConnected() ){
pyExec("import os")
os <- pyObject("os", regFinalizer = FALSE)
ls(os)
## To show again the difference between pyGet and pyGet0.
os1 <- pyGet0("os") ## has no finalizer
os2 <- pyGet("os")  ## has a finalizer
os$py.variableName
os1$py.variableName
os2$py.variableName
}
```

`pyOptions` *Options for the PythonInR package*

Description

a function for getting and setting options in the PythonInR package.

Usage

```
pyOptions(option, value)
```

Arguments

`option`	a character giving the option to set or get.
`value`	the new value of the option.

Details

The following options are available:

- **numpyAlias** a character giving the numpy alias, the default value is "numpy".
- **useNumpy** a logical giving if numpy should be used if getting and setting matrices.
- **pandasAlias** a character giving the pandas alias, the default value is "pandas".
- **usePandas** a logical giving if pandas should be used if getting and setting data.frames.
- **winPython364** a logical indicating if Python 3 64-bit under windows is used, this option is set automatically at startup and shouldn't be changed.

Examples

```
## Not run:
pyOptions()
pyExec("import numpy as np")
pyOptions("numpyAlias", "np")
pyOptions("useNumpy", TRUE)
pyExec("import pandas as pd")
pyOptions("pandasAlias", "pd")
pyOptions("usePandas", TRUE)

## End(Not run)
```

pyPrint *Convenience function to print a given Python object to the R terminal*

Description

Prints a given Python variable to the R terminal.

Usage

```
pyPrint(objName)
```

Arguments

objName	a character string to be evaluated in Python and printed to the R terminal.

Details

Internally it uses a combination of pyExec and print. Please note that the result of pyExecp("x") and pyPrint("x") often will be different, since pyExecp("x") is equivalent to typing x into the Python terminal whereas pyPrint("x") is equivalent to typing print(x) into the Python terminal.

Examples

```
pyPrint("'Hello ' + 'R!'")
pyPrint("sys.version")
```

pySet *assigns R objects to Python*

Description

The function pySet allows to assign R objects to the Python namespace, the conversion from R to Python is done automatically.

Usage

```
pySet(key, value, namespace = "__main__", useSetPoly = TRUE,
  useNumpy = pyOptions("useNumpy"), usePandas = pyOptions("usePandas"))
```

Arguments

key	a string specifying the name of the Python object.
value	a R object which is assigned to Python.
namespace	a string specifying where the key should be located. If the namespace is set to "__main__" the key will be set to the global namespace. But it is also possible to set attributes of objects e.g. the attribute name of the object 'os'.
useSetPoly	an optional logical, giving if pySetPoly should be used to transform R objects into Python objects. For example if useSetPoly is TRUE unnamed vectors are transformed to Python objects of type PrVector else to lists.
useNumpy	an optional logical, default is FALSE, to control if numpy should be used for the type conversion of matrices.
usePandas	an optional logical, default is FALSE, to control if pandas should be used for the type conversion of data frames.

Details

More information about the type conversion can be found in the README file or at http://pythoninr.bitbucket.org/.

Examples

```
pySet("x", 3)
pySet("M", diag(1,3))
pyImport("os")
pySet("name", "Hello os!", namespace="os")
## In some situations it can be beneficial to convert R lists or vectors
## to Python tuple instead of lists. One way to accomplish this is to change
## the class of the vector to "tuple".
y <- c(1, 2, 3)
class(y) <- "tuple"
pySet("y", y)
## pySet can also be used to change values of objects or dictionaries.
asTuple <- function(x) {
 class(x) <- "tuple"
 return(x)
}
pyExec("d = dict()")
pySet("myTuple", asTuple(1:10), namespace="d")
pySet("myList", as.list(1:5), namespace="d")
```

---

pyTuple *Creates a virtual Python tuple*

Description

The function pyTuple creates a virtual Python object of type PythonInR_Tuple.

Usage

```
pyTuple(key, value, regFinalizer = FALSE)
```

Arguments

key	a character string giving the name of the Python object.
value	an optional value, allowed values are vectors, lists and NULL.
regFinalizer	a logical indicating if a finalizer should be be registered, the default value is TRUE.

Details

If no value is provided a virtual Python tuple for an existing Python object is created. If the value is NULL an empty virtual Python object for an empty tuple is created. If the value is a vector or tuple a new Python object based on the vector or list is created.

Examples

```
if ( pyIsConnected() ){
pyExec('myPyTuple = (1, 2, 5, "Hello R!")')
# create a virtual Python tuple for an existing tuple
myTuple <- pyTuple("myPyTuple")
myTuple[0]
tryCatch({myTuple[1] <- "should give an error since tuple are not mutable"},
         error = function(e) print(e))
myTuple
# create a new Python tuple and virtual tuple
newTuple <- pyTuple('myNewTuple', list(1:3, 'Hello Python'))
newTuple[1]
}
```

---

pyType — *Convenience function to call the Python function* **type**

Description

Convenience function to call the Python function **type**.

Usage

```
pyType(objName)
```

Arguments

objName	a string specifying the name of the Python object.

Value

The type of the specified object as character on success, NULL otherwise.

Examples

```
pyExec("x = dict()")
pyType("x")
```

`pyVersion` *is a convenience function to get sys.version from Python*

Description

A convenience function to get sys.version.

Usage

```
pyVersion()
```

Value

Returns a string containing the Python version and some compiler information.

Examples

```
pyVersion()
```

Index