r: the instance of R

This class is currently a singleton, with its one representation instanciated when the module is loaded:

```
>>> robjects.r
>>> print(robjects.r)
```

The instance can be seen as the entry point to an embedded R process.

Being a singleton means that each time the constructor for `R` is called the same instance is returned; this is required by the fact that the embedded R is stateful.

The elements that would be accessible from an equivalent R environment are accessible as attributes of the instance. Readers familiar with the `ctypes` module for Python will note the similarity with it.

R vectors:

```
>>> pi = robjects.r.pi
>>> letters = robjects.r.letters
```

R functions:

```
>>> plot = robjects.r.plot
>>> dir = robjects.r.dir
```

This approach has limitation as:

- The actual Python attributes for the object masks the R elements
- **'.' (dot) is syntactically valid in names for R objects, but not for**
  python objects.

That last limitation can partly be removed by using `rpy2.rpy_classic` if this feature matters most to you.

```
>>> robjects.r.as_null
# AttributeError raised
>>> import rpy2.rpy_classic as rpy
>>> rpy.set_default_mode(rpy.NO_CONVERSION)
>>> rpy.r.as_null
# R function as.null() returned
```

Note

The section Partial use of rpy_classic outlines how to integrate `rpy2.rpy_classic` code.

Behind the scene, the steps for getting an attribute of *r* are rather straightforward:

1. Check if the attribute is defined as such in the python definition for *r*
2. Check if the attribute is can be accessed in R, starting from *globalenv*

When safety matters most, we recommend using `__getitem__()` to get a given R object.

```
>>> as_null = robjects.r['as.null']
```

Storing the object in a python variable will protect it from garbage collection, even if deleted from the objects visible to an R user.

```
>>> robjects.globalenv['foo'] = 1.2
>>> foo = robjects.r['foo']
>>> foo[0]
1.2
```

Here we *remove* the symbol *foo* from the R Global Environment.

```
>>> robjects.r['rm']('foo')
>>> robjects.r['foo']
LookupError: 'foo' not found
```

The object itself remains available, and protected from R's garbage collection until *foo* is deleted from Python

```
>>> foo[0]
1.2
```

Evaluating a string as R code

Just like it is the case with RPy-1.x, on-the-fly evaluation of R code contained in a string can be performed by calling the *r* instance:

```
>>> print(robjects.r('1+2'))
[1] 3
>>> sqr = robjects.r('function(x) x^2')
```

```
>>> print(sqr)
function (x)
x^2
>>> print(sqr(2))
[1] 4
```

The astute reader will quickly realize that R objects named by python variables can be plugged into code through their **R** representation:

```
>>> x = robjects.r.rnorm(100)
>>> robjects.r('hist(%s, xlab="x", main="hist(x)")' %x.r_repr())
```

Warning

Doing this with large objects might not be the best use of your computing power.