Package: oor (via r-universe)

September 14, 2024

Title Object Oriented R

Version 0.0.1

Description Lightweight tools for building object-oriented R code.

License GPL (>= 3)

Encoding UTF-8

LazyData true

Roxygen list(markdown = TRUE)

RoxygenNote 7.2.1

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

RemoteUrl https://github.com/canmod/oor

RemoteRef HEAD

RemoteSha 6bbd032f12286e17ceee82106b5d2e8feaab8690

Contents

`Base` *Base Class*

Description

Initialize an empty object.

Usage

```
Base(starting_environment = emptyenv())
```

Arguments

`starting_environment`
: An environment to enclose the empty object. This enclosing environment is useful for making things other than `self` available to methods. I have found that it is usually best to ignore this possibility, but it might indeed be useful from time-to-time.

Details

Inherit from `Base` if you want to start from an empty class

Examples

```
empty_object = Base()
print(empty_object)
names(empty_object)

Printer = function(x) {
  self = Base()
  self$.x = x
  self$print = function() print(self$.x)
```

```
  return_object(self, "Printer")
}
printer = Printer("something to print")
printer$print()

SupportivePrinter = function(x) {
  self = Printer(x)
  self$print = function() {
    print(paste(sQuote(self$.x), "is a very nice thing to say"))
  }
  return_object(self, "Supportive")
}
supportive_printer = SupportivePrinter("something to print")
supportive_printer$print()
```

`clean_method_environment` *Clean Method Environment*

Description

Clean the environment of a method (or methods) so that they contain a single object – self – which is the environment defining an object

Usage

```
clean_method_environment(e)
```

Arguments

`e`	Environment containing methods in an object

Value

There is no return value. The function is called for its side-effect of cleaning a method environment.

`Implementation` *Implementation Class*

Description

Initialize an object with concrete implementations of abstract method definitions.

Usage

```
Implementation()
```

Details

Inherit from `Implementation` (using the `implements` utility function) if you want your class to implement an `Interface`.

Examples

```
BinaryOperation = function() {
  self = Interface()
  self$operate = function(x = numeric(1L), y = numeric(1L)) return(numeric(1L))
  return_object(self, "BinaryOperation")
}
Add = function() {
  self = implements(BinaryOperation)
  self$operate = function(x = numeric(1L), y = numeric(1L)) return(x + y)
  return_object(self, "Add")
}
Multiply = function() {
  self = implements(BinaryOperation)
  self$operate = function(x = numeric(1L), y = numeric(1L)) return(x * y)
  return_object(self, "Multiply")
}
Add()$operate(1, 1)
Multiply()$operate(2, 2)
```

`inheritance` *Inheritance*

Description

Inherit methods and fields from other classes.

Usage

```
inherit_from(parent, traits, ...)

implements(interface)
```

Arguments

`parent`	A single class from which to inherit methods and fields.
`traits`	A vector of `Trait` classes from which to forward methods (trait classes are like mixin class in Python).
`...`	Arguments to pass to the initialization of the `parent` class.
`interface`	Class definition that inherits from `Interface`

Details

There are three ways to inherit from other classes: (1) directly (2) using the `inherit_from` function (3) using the `implements` function

Each of these ways works by adding a line that creates an object called `self` at the beginning of a class definition. The object `self` is an object of class `ParentClass` and you are free to add new fields and methods to this object.

Direct Inheritance

```
self = ParentClass(...)
```

Here `ParentClass` is the name of the parent class being inherited from. In most cases, direct inheritance is the most useful approach. The other two are for more advanced use.

Inherit From

```
self = inherit_from(ParentClass, list_of_trait_classes)
```

Here `ParentClass` is the class being directly inherited from and `list_of_trait_classes` is a list of `Trait` classes containing methods to be forwarded to `self`.

Implementations

```
self = implements(ParentInterface)
```

Here `ParentInterface` is an abstract set of method signatures. Following this initialization, concrete definitions of these abstract method need to be added to `self`. This process is referred to as implementing an interface.

`Interface` *Interface Class*

Description

Initialize an empty abstract class.

Usage

```
Interface()
```

Details

Inherit from `Interface` to define the argument signatures and return value types of abstract methods.

Examples

```
BinaryOperation = function() {
  self = Interface()
  self$operate = function(x = numeric(1L), y = numeric(1L)) return(numeric(1L))
  return_object(self, "BinaryOperation")
}
```

`Is` *Test inheritance*

Description

Test inheritance

Usage

```
Is(class)
```

Arguments

`class`	Name of a class to test for.

`MappedAllTest` *Mapped All Test*

Description

Test that all `MappedTest` results are `TRUE`

Usage

```
MappedAllTest(basic_tester)
```

Arguments

`basic_tester`	An object that can be converted to a function

MappedAnyTest *Mapped Any Test*

Description

Test that any MappedTest results are TRUE

Usage

```
MappedAnyTest(basic_tester)
```

Arguments

basic_tester	An object that can be converted to a function

MappedSummarizer *Mapped Summarizer*

Description

Apply a Summarizer to each element of a list, in order to test that a particular summary of each lists item meets a certain criterion. MappedSummarizers are typically included in TestPipelines.

Usage

```
MappedSummarizer(...)
```

Arguments

...	A list of summarizing functions.

Value

Object of class Test that summarizes each element of objects to test.

MappedTest *Mapped Test*

Description

Apply a Test to each element of a list.

Usage

```
MappedTest(basic_tester, boolean_aggregator)
```

Arguments

basic_tester	An object that can be converted to a function
boolean_aggregator	A function that summarizes a logical vector.

method_apply *Method Apply*

Description

Call a method for each item in a list of objects.

Usage

```
method_apply(objects, method_name, ...)
```

Arguments

objects	List of objects.
method_name	Character string giving the name of the method.
...	Arguments to pass to the method.

MultiTest *Multi Test*

Description

Assess several criteria.

Usage

```
MultiTest(test_function_list, boolean_aggregator)

All(...)

Any(...)
```

Arguments

test_function_list	List of objects of class Test or function.
boolean_aggregator	A function that summarizes a logical vector.
...	Test functions.

Value

Object of class Test that tests several criteria at the same time.

Functions

- All(): Test that all of the criteria are met.
- Any(): Test that any of the criteria are met.

Examples

```
is_matrix = All(
 is.numeric,
 TestPipeline(
   Summarizer(dim, length),
   TestRange(0, 2)
 )
)
is_matrix$apply(array("a", c(1))) # FALSE
is_matrix$apply(array("a", c(1, 1, 2))) # FALSE
is_matrix$apply(array(1, c(1, 1, 2))) # FALSE
is_matrix$apply(array(1, c(1, 2))) # TRUE
is_matrix$apply(1) # TRUE
```

Not *Not*

Description

Not

Usage

```
Not(basic_tester)
```

Arguments

basic_tester	An object that can be converted to a function

Value

Object of class Test that evaluates the complement of basic_tester.

Examples

```
Not(is.numeric)$apply(1) # FALSE
Not(is.numeric)$apply("1") # TRUE
```

return_facade *Return Facade*

Description

Experimental

Usage

```
return_facade(self, private, class)
```

Arguments

self	New object.
private	Environment to use for containing private methods and fields.
class	String giving the class name.

return_object *Return Object*

Description

This should be the final function called in a class definition. Think of it like return(...)

Usage

```
return_object(self, class)
```

Arguments

self	New object.
class	String giving the class name.

Value

New object of class given by class.

Summarizer *Summarizer*

Description

Summarize an object to be tested, so that the test is applied to the summary and not the object itself (e.g. length(dim(object)) == 2L). Summarizers are typically included in TestPipelines.

Usage

```
Summarizer(...)
```

Arguments

...	A list of summarizing functions.

Value

Object of class Test that summarizes objects to test.

Test *Abstract Class Testing Objects*

Description

Abstract Class Testing Objects

Usage

```
Test()

## S3 method for class 'Test'
as.function(x, ...)
```

Arguments

x	Test object to convert to a function
...	Not used. Present for S3 method consistency.

Value

Object with an apply method that takes a single argument, x, and returns a length-one logical vector.

Testable *Testable Class*

Description

Initialize an object with functionality for validating objects

Usage

```
Testable()
```

Details

Inherit from Testable if you would like your class to provide a validity check. Validity checking often requires differentiating between public versus private members, as well as fields versus methods.

Examples

```
Printer = function(x) {
  self = Testable()
  self$.x = x
  self$valid = function() {
     if (!is.character(self$.x)) {
       return("can only print character strings")
     }
     if (length(self$.x) != 1L) {
       return("can only print length-1 character vectors")
     }
     return(TRUE)
  }
  self$print = function() print(self$.x)
  return_object(self, "Printer")
}
printer = Printer("something to print")
printer$print()
try(Printer(0)) ## error
```

`TestBasic` *Basic Test*

Description

Basic Test

Usage

```
TestBasic(basic_tester)
```

Arguments

`basic_tester`	An object that can be converted to a function

Value

Object of class `Test` that evaluates the `basic_tester`.

TestFalse *Test False*

Description

Test False

Usage

```
TestFalse()
```

TestHomo *Test for Homogeneity*

Description

Test that all elements in an object are identical.

Usage

```
TestHomo()
```

Value

Object of class `Test` that tests that all elements in an object are identical.

TestPipeline *Test Pipeline*

Description

Test Pipeline

Usage

```
TestPipeline(...)
```

Arguments

`...`	Objects of class `Test` or `function`. The final object in this list must be either a `Test` object with an `apply` method that returns a length-one logical vector or a function that does so.

Value

Object inheriting from Test

Examples

```
is_matrix = TestPipeline(
  Summarizer(dim, length),
  TestRange(0, 2)
)
is_matrix$apply(array("a", c(1))) # TRUE
is_matrix$apply(array(1, c(1, 2, 3))) # FALSE

each_is_matrix = TestPipeline(
  MappedSummarizer(dim, length),
  All(TestRange(0, 2))
)
each_is_matrix$apply(list(1, matrix(1, 2, 3), "a")) # TRUE
each_is_matrix$apply(list(1, array(1, c(2, 3, 4)), "a")) # FALSE
```

TestPlaceholder *Placeholder for a Test*

Description

Always return TRUE.

Usage

```
TestPlaceholder()
```

TestRange *Range Test*

Description

Test that all elements in an object greater than or equal to `lower` and less than or equal to `upper`.

Usage

```
TestRange(lower, upper)
```

Arguments

`lower`	Lower bound
`upper`	Upper bound

Value

Object of class `Test` that tests that all elements in an object numerically on a particular range.

TestSubset *Subset Test*

Description

Test that all elements in an object are in `set`

Usage

```
TestSubset(set)
```

Arguments

`set`	Universe of possibilities.

Value

Object of class `Test` that tests that all elements in an object are in a particular set.

TestTrue *Test True*

Description

Test True

Usage

```
TestTrue()
```

Trait *Trait Class*

Description

Initialize an object with methods that are intended to be forwarded to other classes.

Usage

```
Trait()
```

Details

Inherit from `Trait` if you want to use your class to forward public methods to other classes without direct inheritance.

Examples

```
Print = function(x) {
  self = Testable()
  self$.x = x
  return_object(self, "Print")
}
Printer = function() {
  self = Trait()
  self$print = function() print(self$.x)
  return_object(self, "Printer")
}
PrintString = function(x) {
  self = inherit_from(Print, list(Printer), x)
  self$valid = function() {
     if (!is.character(self$.x)) return("can only print character strings")
     if (length(self$.x) != 1L) return("can only print length-1 character vectors")
     return(TRUE)
  }
  return_object(self, "PrintString")
}
PrintNumber = function(x) {
  self = inherit_from(Print, list(Printer), x)
  self$valid = function() {
     if (!is.numeric(self$.x)) return("can only print character strings")
     return(TRUE)
  }
  return_object(self, "PrintNumber")
}
PrintString("something to print")$print()
PrintNumber(pi)$print()
try(PrintNumber("not a number")) ## error
```

Unclean *Unclean*

Description

Experimental

Usage

```
Unclean()
```

validate_object *Validate Object*

Description

S3 generic for checking the validity of a constructed object. should either return nothing or trigger an error.

Usage

```
validate_object(object)
```

Arguments

object	Object to be validated.

Value

TODO – check $valid methods

ValidityMessager *Validity Messager*

Description

Couple a test function with a failure message

Usage

```
ValidityMessager(test_function, ...)
```

Arguments

test_function	Object that is coercible to a function, typically a function that will return a length-1 logical vector. Often this object will inherit from Test, which provides a way to compose object tests.
...	Length-1 character vectors to display if test_function does not return TRUE.

Details

ValidityMessager objects have an assert method with one argument, x. If the test function evaluates to TRUE then the argument, x, is returned. If it does not return TRUE then the failure message is given.

Value

Object of class ValidityMessager containing a check method that will return TRUE or fail with fail_message.

Examples

```
is_numeric = ValidityMessager(is.numeric, "not numeric")

try(is_numeric$check("1"))

HoldANumber = function(x) {
  self = Base()
  self$x =  is_numeric$assert(x)
  return_object(self, "HoldANumber")
}
try(HoldANumber("a")) ## error message
HoldANumber(1) ## success
```

Index