Package 'unitizer'

May 18, 2024

Title Interactive R Unit Tests

Description Simplifies regression tests by comparing objects produced by test code with earlier versions of those same objects. If objects are unchanged the tests pass, otherwise execution stops with error details. If in interactive mode, tests can be reviewed through the provided interactive environment.

Version 1.4.21

Depends methods

Imports stats, utils, crayon (>= 1.3.2), diffobj (>= 0.1.5.9000)

VignetteBuilder knitr

Suggests knitr, rmarkdown

License GPL-2 | GPL-3

URL https://github.com/brodieG/unitizer

BugReports https://github.com/brodieG/unitizer/issues

Collate 'asciiml.R' 'capture.R' 'is.R' 'global.R' 'change.R' 'class_unions.R' 'list.R' 'conditions.R' 'item.R' 'deparse.R' 'text.R' 'item.sub.R' 'section.R' 'test.R' 'unitizer.R' 'exec.R' 'prompt.R' 'browse.struct.R' 'browse.R' 'demo.R' 'diff.R' 'faux_prompt.R' 'get.R' 'heal.R' 'load.R' 'ls.R' 'misc.R' 'search.R' 'options.R' 'onload.R' 'parse.R' 'rename.R' 'repairenvs.R' 'result.R' 'shims.R' 'size.R' 'state.R' 'state.compare.R' 'traceback.R' 'translate.R' 'unitize.R' 'unitize.core.R' 'unitizer-package.R' 'unitizer.add.R' 'upgrade.R'

RoxygenNote 7.2.3

NeedsCompilation no

Author Brodie Gaslam [aut, cre],
R Core Team [cph] (Traceback function sources.)

Maintainer Brodie Gaslam <brodie.gaslam@yahoo.com>

Repository CRAN

Date/Publication 2024-05-18 19:50:03 UTC

R topics documented:

`all.equal.condition` *Compare Conditions*

Description

Tests that issue warnings or 'stop' produce `condition` objects. The functions documented here are specialized versions of `all.equal` designed specifically to compare conditions and condition lists produced during `unitizer` test evaluations. `conditionList` objects are lists of conditions that come about when test expressions emit multiple conditions (e.g. more than one warning).

Usage

```
## S4 method for signature 'conditionList,ANY'
all.equal(target, current, ...)

## S3 method for class 'equal.conditionList'
all(target, current, ...)

## S3 method for class 'equal.condition'
all(target, current, ...)
```

Arguments

target	the list of conditions that we are matching against
current	the list of conditions we are checking
...	provided for compatibility with generic

Details

condition objects produced by tests have one additional attributed "printed" which disambiguates whether a condition was the result of the test expression, or the print / show method used to display it to screen.

For conditionList objects, these methods only return TRUE if all conditions are pairwise all.equal.

Value

TRUE if the (lists of) conditions are equivalent, a character vector explaining why they are not otherwise

Examples

```
cond.1 <- simpleWarning('hello world')
cond.2 <- simpleError('hello world')
cond.3 <- simpleError('goodbye world')
all.equal(cond.1, cond.1)
all.equal(cond.1, cond.2)
all.equal(cond.2, cond.3)
## Normally you would never actually create a `conditionList` yourself; these
## are automatically generated by `unitizer` for review at the `unitizer`
## prompt
all.equal(
  conditionList(.items=list(cond.1, cond.2)),
  conditionList(.items=list(cond.1, cond.3))
)
```

---

all_eq *Like all.equal but Returns Empty String If Not all.equal*

Description

Used as the default value comparison function since when values mismatch we use diffObj which would make the text output from all.equal somewhat redundant.

Usage

```
all_eq(target, current, ...)
```

Arguments

target	R object
current	other R object to be compared to target
...	arguments to pass to all.equal

Value

TRUE if all.equal returns TRUE, "" otherwise all_eq(1, 1L) all_eq(1, 2) isTRUE(all_eq(1, 2))

---

conditionList *Contains A List of Conditions*

Description

Condition lists are S4 classes that contain condition objects emitted by unitizer tests. Condition lists will typically be accessible via the .NEW and .REF unitizer test objects. You can access individual conditions using [[(see examples), and for the most part you can treat them as you would an S3 list containing conditions.

Details

There are show and all.equal methods implemented for them, the latter of which is used to compare conditions across tests. If you wish to implement a custom comparison function via unitizer_sect, your function will need to compare conditionList objects.

Slots

.items list of conditions

Note

Implemented as an S4 class to avoid setOldClass and related compatibility issues; the conditionList class contains unitizerList.

See Also

unitizer_sect, unitizerList, all.equal.conditionList

Examples

```
## Create a test item as you would find normally at the `unitizer` prompt
## for illustrative purposes:
.NEW <- mock_item()
## Access the first condition from the new test evaluation
.NEW$conditions[[1L]]
## loop through all conditions
for(i in seq_along(.NEW$conditions)) .NEW$conditions[[i]]
```

desc *One Line Description of Object*

Description

Objects are described by class, and dimensions. Dimensions is always denoted in square brackets. For example, "int[10]" means an integer of length ten. Typically an object will be identified by `head(class(obj), 1L)` along with its dimensions. Recursive objects will have the first level shown provided that doing so fits within `limit`.

Usage

```
desc(val, limit = getOption("width"))
```

Arguments

`val`	object to describe
`limit`	max characters to display

Details

Eventually this will be migrated to an S3 generic to allow recursive dispatch on object type.

Value

character(1L) describing object

Examples

```
desc(list(a=iris, b=lm(dist ~ speed, cars), 1:10, matrix(letters, 2)))
```

editCalls *Edit Calls In Unitizer*

Description

Used if you want to change language in test expression in a unitizer when the actual results of running the expressions is unchanged. This is useful if you decided to rename functions, etc., without having to re-run the entire `unitize` process since `unitize` matches tests based on expressions.

Usage

```
editCalls(x, lang.old, lang.new, ...)

## S4 method for signature 'unitizer,language,language'
editCalls(
  x,
  lang.old,
  lang.new,
  interactive.mode = interactive(),
  interactive.only = TRUE,
  ...
)
```

Arguments

x	a unitizer object
lang.old	the name of the function replace
lang.new	the new name of the function
...	unused
interactive.mode	logical(1L) whether to run in interactive mode (request user input when needed) or not (error if user input is required, e.g. if all tests do not pass).
interactive.only	logical(1L) set to FALSE if you want to allow this to run in non-interactive mode, but warnings will be suppressed and will proceed without prompting, obviously...

Value

a untizer object with function names modifies

Note

this is a somewhat experimental function, so make sure you backup any unitizers before you try to use it.

Examples

```
## Not run:
untz <- get_unitizer("tests/unitizer/mytests.unitizer")
untz.edited <- editCalls(untz, quote(myFun), quote(my_fun))
set_unitizer("tests/unitizer/mytests.unitizer", untz.edited)

## End(Not run)
```

filename_to_storeid *Create a Store ID from a Test File Name*

Description

Create a Store ID from a Test File Name

Usage

```
filename_to_storeid(x)
```

Arguments

x	character(1L) file name ending in .r or .R

Value

store id name, or NULL if x doesn't meet expectations

Examples

```
filename_to_storeid(file.path("tests", "unitizer", "foo.R"))
filename_to_storeid(file.path("tests", "unitizer", "boo.r"))
# does not end in [rR]
filename_to_storeid(file.path("tests", "unitizer", "boo"))
```

healEnvs *Fix Environment Ancestries*

Description

This is an internal method and exposed so that this aspect of unitizer is documented for package users (see Details).

Usage

```
## S4 method for signature 'unitizerItems,unitizer'
healEnvs(x, y, ...)
```

Arguments

x	unitizerItems object
y	unitizer object x was generated from
...	unused, here for inheriting methods

Details

Environment healing is necessary because when we let the user pick and chose which tests to store and which ones to reject, there may no longer be a clear ancestry chain within the remaining tests.

The healing process is somewhat complex and full of compromises. We are attempting to create a self consistent set of nested parent environments for each test, but at the same time, we don't want to store all the combinations of reference and new objects.

We only store new objects in `unitizer`, with the lone exception of objects associated to a test environment. These will include any assignments that occur just prior to a test, as well as any objects created by the actual test.

There are two ways in which we modify the environment ancestry. If the user decides to not store some new tests, then the objects created in between the previous new stored test and the next new stored test are all moved to the next new stored test, and the previous new stored test becomes the parent of the next new stored test.

The second way relates to when the user decides to keep a reference test over a matching new test. This is a lot more complicated because we do not preserve the reference test environment ancestry. Effectively, we need to graft the reference test to the new environment ancestry.

If a reference test that is being kept matches directly to a new test, then the parent of that new test becomes the parent of the reference test.

If there is no direct match, but there are child reference tests that match to a new item, then the parent is the youngest new test that is older than the new test that was matched and is kept. If no new tests meet this criterion, then base.env is the parent.

If there is no direct match, and there are no child reference tests that are being kept that do match to a kept new item, then the parent will be the last new test that is kept.

The main takeaway from all this is that reference tests don't really keep their evaluation environment. Often this environment is similar to the new environment. When there are difference between the two, the output of `ls` is customized to highlight which objects were actually available/unmodified at the time of the reference test evaluation. Object names will have the following symbols appended to explain the object status:

- ': object exists in browsing environment, but not the same as it was when test was evalaluated
- *: object was present during test evaluation but is not available in unitizer anymore
- **: object was not present during test evaluation, but exists in current environment

Value

`unitizerItems`

Note

Could be more robust by ensuring that items in `x` actually do come from `y`. This is particularly important since when we re-assemble the final list, we don't actually use x at all. Signature for this should probably ultimately change to be something like `c("unitizer", "x")` where x is just a data frame with column 1 the item index, and column 2 whether it originated from "new" or "ref"

See Also

updateLs,unitizerItem-method

---

infer_unitizer_location
Infers Possible Unitizer Path From Context

Description

Used by most unitizer functions that operate on unitizers to make it easy in interactive use to specify the most likely intended unitizer in a package or a directory. For 'R CMD check' and similar testing should not rely on this functionality.

Usage

```
infer_unitizer_location(store.id, ...)

## Default S3 method:
infer_unitizer_location(store.id, ...)

## S3 method for class 'character'
infer_unitizer_location(
  store.id,
  type = "f",
  interactive.mode = interactive(),
  ...
)
```

Arguments

store.id	character(1L) file or directory name, the file name portion (i.e after the last slash) may be partially specified
...	arguments to pass on to other methods
type	character(1L) in c("f", "u", "d"), "f" for test file, "d" for a directory, "u" for a unitizer directory
interactive.mode	logical(1L) whether to allow user input to resolve ambiguities

Details

This is implemented as an S3 generic to allow third parties to define inference methods for other types of store.id, but the documentation here is for the "character" method which is what unitizer uses by default.

If store.id is a directory that appears to be an R package (contains DESCRIPTION, an R folder, a tests folder), will look for candidate files in file.path(store.id, "tests", "unitizer"), starting with files with the same name as the package (ending in ".R" or ".unitizer" if type is "f"

or "u" respectively), or if there is only one file, that file, or if there are multiple candidate files and in interactive mode prompting user for a selection. If type is "d", then will just provide the "tests/unitizer" directory.

If name is not a directory, will try to find a file by that name, and if that fails, will try to partially match a file by that name. Partial matching requires the front portion of the name to be fully specified and no extension be provided (e.g. for "mytests.R", "myt" is valid, but "tests" and "myt.R" are both invalid). Partially specified files may be specified in subdirectories (e.g. "tests/myt").

Inference assumes your files end in ".R" for code files and ".unitizer" for unitizer data directories.

If store.id is NULL, the default infer_unitizer_location method will attempt to find the top level package directory and then call the character method with that directory as store.id. If the parent package directory cannot be found, then the character method is called with the current directory as the argument.

Value

character(1L) an inferred path, or store.id with a warning if path cannot be inferred

See Also

get_unitizer for discussion of alternate store.id objects

---

mock_item *Generates a Dummy Item For Use in Examples*

Description

The only purpose of this function is to create a unitizerItem for use by examples.

Usage

```
mock_item()
```

Value

unitizerItem object

repair_environments *Repair Environment Chains*

Description

In theory should never be needed, but use in case you get errors about corrupted environments. You should only use this if you get an error telling you to use it.

Usage

```
repair_environments(x, interactive.mode = interactive())
```

Arguments

x	either a unitizer, or a store id (see unitize)
interactive.mode	logical(1L) whether to run in interactive mode (request user input when needed) or not (error if user input is required, e.g. if all tests do not pass).

Details

If you pass a store id this will re-save the repaired unitizer to the location specified by the store id.

Value

a unitizer object

See Also

unitize

set_unitizer *Set and Retrieve Store Contents*

Description

These functions are not used directly; rather, they are used by unitize to get and set the unitizer objects. You should only need to understand these functions if you are looking to implement a special storage mechanism for the unitizer objects.

Usage

```
set_unitizer(store.id, unitizer)

get_unitizer(store.id)

## S3 method for class 'character'
get_unitizer(store.id)

## Default S3 method:
get_unitizer(store.id)

## S3 method for class 'unitizer_result'
get_unitizer(store.id)

## S3 method for class 'unitizer_results'
get_unitizer(store.id)
```

Arguments

`store.id`	a filesystem path to the store (an .rds file)
`unitizer`	a `unitizer-class` object containing the store data

Details

By default, only a character method is defined, which will interpret its inputs as a filesystem path to the `unitizer` folder. RDSes of serialization type 2 will be stored and retrieved from there. The serialization format may change in the future, but if R maintains facilities to read/write type 2, we will provide the option to use that format. At this time there is no API to change the serialization format.

You may write your own methods for special storage situations (e.g SQL database, ftp server, etc) with the understanding that the getting method may only accept one argument, the `store.id`, and the setting method only two arguments, the `store.id` and the `unitizer`.

S3 dispatch will be on `store.id`, and `store.id` may be any R object that identifies the unitizer. For example, a potential SQL implementation where the unitizers get stored in blobs may look like so:

```
my.sql.store.id <- structure(
  list(
    server="myunitizerserver.mydomain.com:3306",
    database="unitizers",
    table="project1",
    id="cornercasetests"
  ),
  class="sql_unitizer"
)
get_unitizer.sql_unitizer <- function(store.id) { # FUNCTION BODY }
set_unitizer.sql_unitizer <- function(store.id, unitizer) { # FUNCTION BODY }
```

```
unitize("unitizer/cornertestcases.R", my.sql.store.id)
```

Make sure you also define an `as.character` method for your object to produce a human readable identifying string.

For inspirations for the bodies of the _store functions look at the source code for `unitizer:::get_unitizer.character` and `unitizer:::set_unitizer.character`. Expectations for the functions are as follows. `get_unitizer` must:

- return a `unitizer-class` object if `store.id` exists and contains a valid object
- return FALSE if the object doesn't exist (e.g. first time run-through, so reference copy doesn't exist yet)
- `stop` on error

`set_unitizer` must:

- return TRUE on success
- `stop` on error

Value

- set_unitizer TRUE if unitizer storing worked, error otherwise
- get_unitizer a `unitizer-class` object, FALSE if `store.id` doesn't exist yet, or error otherwise; note that the `unitizer_results` method returns a list

See Also

`saveRDS`

---

show.conditionList *Prints A list of Conditions*

Description

S4 method for `conditionList` objects.

Usage

```
## S4 method for signature 'conditionList'
show(object)
```

Arguments

`object`	a `conditionList` object (list of conditions)

Value

object, invisibly

See Also

conditionList

Examples

```
## Create a test item as you would find normally at the `unitizer` prompt
## for illustrative purposes:
.NEW <- mock_item()
## Show the conditions the test generated (typing `show` here is optional
## since auto-printing should dispatch to `show`)
show(.NEW$conditions)
```

---

testFuns *Store Functions for New vs. Reference Test Comparisons*

Description

testFuns contains the functions used to compare the results and side effects of running test expressions. "testFuns" objects can be used as the compare argument for unitizer_sect, thereby allowing you to specify different comparison functions for different aspects of test evaluation.

Details

The default comparison functions are as follows:

- value: all_eq
- conditions: all_eq
- output: function(x, y) TRUE, i.e. not compared
- message: function(x, y) TRUE, i.e. not compared as conditions should be capturing warnings/errors
- aborted: function(x, y) TRUE, i.e. not compared as conditions should also be capturing this implicitly

See Also

unitizer_sect for more relevant usage examples, all_eq

Examples

```
# use `identical` instead of `all.equal` to compare values
testFuns(value=identical)
```

testthat_translate_file *Convert a* testthat *Test File to a* unitizer

Description

Converts a **copy** of an existing testthat test file to a unitizer test file and test store, or a directory of such files to a corresponding unitizer directory. See examples.

Usage

```
testthat_translate_file(
  file.name,
  target.dir = file.path(dirname(file.name), "..", "unitizer"),
  state = getOption("unitizer.state"),
  keep.testthat.call = TRUE,
  prompt = "always",
  interactive.mode = interactive(),
  use.sects = TRUE,
  unitize = TRUE,
  ...
)

testthat_translate_dir(
  dir.name,
  target.dir = file.path(dir.name, "..", "unitizer"),
  filter = "^test.*\\.[rR]",
  state = getOption("unitizer.state"),
  keep.testthat.call = TRUE,
  force = FALSE,
  interactive.mode = interactive(),
  use.sects = TRUE,
  unitize = TRUE,
  ...
)

testthat_translate_name(
  file.name,
  target.dir = file.path(dirname(file.name), "..", "unitizer"),
  name.new = NULL,
  name.pattern = "^(?:test\\W*)?(.*)(?:\\.[rR])$",
  name.replace = "\\1"
)
```

Arguments

file.name	a path to the testthat test file to convert

target.dir — the directory to create the unitizer test file and test store in; for testthat_translate_file only: if NULL will return as a character vector what the contents of the translated file would have been instead of writing the file

state — what state control to use (see same argument for unitize)

keep.testthat.call — whether to preserve the testthat call that was converted, as a comment

prompt — character(1L):

- "always" to always prompt before writing new files
- "overwrite" only prompt if existing file is about to be overwritten
- "never" never prompt

interactive.mode — logical(1L) primarily for testing purposes, allows us to force prompting in non-interactive mode; note that unitize and unitize_dir are always called in non-interactive mode by these functions, this parameter only controls prompts generated directly by these functions.

use.sects — TRUE (default) or FALSE whether to translate test_that sections to unitizer_sect or simply to turn them into comment banners.

unitize — TRUE (default) or FALSE whether to run unitize after the files are translated.

... — params to pass on to testthat_translate_name

dir.name — a path to the testthat directory to convert

filter — regular expression to select what files in a director are translated

force — logical(1L) whether to allow writing to a target.dir that contains files (implies prompt="never" when testthat_translate_dir runs testthat_translate_file)

name.new — character(1L) the base name for the unitizer files; do not include an extension as we will add it (".R" for the testfile, ".unitizer" for the data directory); set to NULL to generate the name from the testthat file name

name.pattern — character(1L) a regular expression intended to match the testthat test file name (see name.replace) if name.pattern matches, then the new file name will be constructed with this (used as replace parameter to sub); in addition we will add ".R" and ".unitizer" as the extensions for the new files so do not include extensions in your name.replace parameter

name.replace — character(1L) the replacement token, typically would include a "\1" token that is filled in by the match group from name.pattern

Value

a file path or a character vector (see target.dir)

Disclaimers

If you already have an extensive test suite in testthat and you do not intend to modify your tests or code very much there is little benefit (and likely some drawbacks) to migrating your tests to unitizer. Please see the introduction vignette for a (biased) view of the pros and cons of unitizer relative to testthat.

These translation functions are provided for your convenience. The unitizer author does not use them very much since he seldom needs to migrate testthat tests. As a result, they have not been tested as thoroughly as the rest of unitizer. Translation is designed to work for the most common testthat use cases, but may not for yours. Make sure you review the resulting unitizers to make sure they contain what you expect before you start relying on them. This is particularly important if your testthat test files are not meant to be run stand-alone with just test_file (see "Differences That May Cause Problems").

Note you can also unitize your testthat files **without** translating them (see notes).

Workflow

1. Start a fresh R session
2. Run your testthat tests with test_dir to ensure they are still passing. If your tests are are runnable only via test_check because they directly access the namespace of your package, see "Differences That May Cause Problems" below
3. Run testthat_dir_translate
4. [optional] use review to review the resulting unitizer(s)

We recommend using testthat_translate_dir over testthat_translate_file because the former also copies and loads any helper files that may be defined. Since libraries used by multiple test files are commonly loaded in these helper files, it is likely that just translating a single file without also copying the helper files will not work properly.

How the Conversion Works

For a subset of the expect_* functions we extract the object parameter and discard the rest of the expectation. For example

```
expect_equal(my_fun(25), 1:10)
```

becomes

```
my_fun(25)
```

. The idea is that on unitizing the expression the result will be output to screen and can be reviewed and accepted. Not all expect_* functions are substituted. For example, expect_is and expect_that are left unchanged because the tests for those functions do not or might not actually test the values of object. expect_gt and similar are also left unchanged as that would require more work than simply extracting the object parameter.

It is perfectly fine to unitize an expect_* call unsubstituted. unitizer captures conditions, values, etc., so if an expect_* test starts failing, it will be detected.

unitizer will then evaluate and store the results of such expressions. Since in theory we just checked our testthat tests were working, presumably the re-evaluated expressions will produce the same values. Please note that the translation process does not actually check this is true (see "Differences That May Cause Problems") so reviewing the results is a good idea.

test_that calls are converted to unitizer_sect calls, and the contents thereof are processed as described above. Calls to context are commented out since there currently is no unitizer

equivalent. Other testthat calls are left unchanged and their return values used as part of the unitizer tests.

Only top level calls are converted. For example, code like for(i in 1:10) expect_equal(my_fun(i), seq(i)) or even (expect_equal(my_fun(10), 1:10)) will not be converted since expect_equal is nested inside a for and (respectively. You will need to manually edit these calls (or just let them remain as is, which is not an issue).

We identify calls to extract based purely on the function symbols (i.e. we do not check whether expect_equal actually resolves to testthat::expect_equal in the context of the test file).

The unitizer files will be created in a sibling folder to the folder containing the testthat files. The names of the new files will be based on the old files. See params target.dir, name.new, name.pattern, and name.replace for more details. We encourage you to try the default settings first as those should work well in most cases.

When using testthat_translate_dir, any files that match "^helper.*[rR]$" are copied over to a '/_pre' subdirectory in "target.dir", and are pre-loaded by default before the tests are unitized.

unitizer Differences That May Cause Problems

If you run your tests during development with test_dir odds are the translation will work just fine. On the other hand, if you rely exclusively on test_check you may need to use state=unitizerStateNoOpt(par.env="pkg when you translate to make sure your tests have access to the internal namespace functions. See unitizerState for details on how to modify state tracking.

If your tests were translated with the state parameter changed from its default value, you will have to use the same value for that parameter in future unitize or unitize_dir runs.

Alternate Use Cases

If you wish to process testthat files for use with the standard R ".Rout" / ".Rout.save process" you can set the unitize and use.sects parameters to FALSE.

See Also

unitize, unitizerState

Examples

```
## Not run:
library(testthat)  # required
testthat_translate_file("tests/testthat/test-random.R")

# Translate `dplyr` tests (assumes `dplyr` source is in './dplyr')
# Normally we would use default `state` value but we cannot in this case
# due to conflicting packages and setup

testthat_translate_dir(
  "dplyr/tests/testthat", state=unitizerStateSafe(par.env="dplyr")
)
# Make sure translation worked (checking one file here)
# *NOTE*: folder we are looking at has changed
```

```
review("dplyr/tests/unitizer/summarise.unitizer")

# Now we can unitize any time we change our code

unitize_dir(
  "dplyr/tests/unitizer", state=unitizerStateSafe(par.env="dplyr")
)

## End(Not run)
```

---

unitize *Unitize an R Test Script*

Description

Turn standard R scripts into unit tests by storing the expressions therein along with the results of their evaluation, and provides an interactive prompt to review tests.

Usage

```
unitize(
  test.file = NULL,
  store.id = NULL,
  state = getOption("unitizer.state"),
  pre = NULL,
  post = NULL,
  history = getOption("unitizer.history.file"),
  interactive.mode = interactive(),
  force.update = FALSE,
  auto.accept = character(0L),
  use.diff = getOption("unitizer.use.diff"),
  show.progress = getOption("unitizer.show.progress", TRUE),
  transcript = getOption("unitizer.transcript", !interactive.mode)
)

review(
  store.id = NULL,
  use.diff = getOption("unitizer.use.diff"),
  show.progress = getOption("unitizer.show.progress", TRUE)
)

unitize_dir(
  test.dir = NULL,
  store.ids = filename_to_storeid,
  pattern = "^[^.].*\\.[Rr]$",
  state = getOption("unitizer.state"),
  pre = NULL,
```

```
  post = NULL,
  history = getOption("unitizer.history.file"),
  interactive.mode = interactive(),
  force.update = FALSE,
  auto.accept = character(0L),
  use.diff = getOption("unitizer.use.diff"),
  show.progress = getOption("unitizer.show.progress", TRUE),
  transcript = getOption("unitizer.transcript", !interactive.mode)
)
```

Arguments

test.file	path to the file containing tests, if supplied path does not match an actual system path, unitizer will try to infer a possible path. If NULL, will look for a file in the "tests/unitizer" package folder if it exists, or in "." if it does not. See infer_unitizer_location) for details.
store.id	if NULL (default), unitizer will select a directory based on the test.file name by replacing .[rR] with .unitizer. You can also specify a directory name, or pass any object that has a defined get_unitizer method which allows you to specify non-standard unitizer storage mechanisms (see get_unitizer). Finally, you can pass an actual unitizer object if you are using review; see store.ids for unitize_dir
state	character(1L) one of c("prisitine", "suggested", "basic", "off", "safe"), an environment, or a state object produced by state or in_pkg; modifies how unitizer manages aspects of session state that could affect test evaluation, including the parent evaluation environment. For more details see unitizerState documentation and vignette("unitizer_reproducible_tests")
pre	NULL, or a character vector pointing to files and/or directories. If a character vector, then any files referenced therein will be sourced, and any directories referenced therein will be scanned non-recursively for visible files ending in ".r" or ".R", which are then also sourced. If NULL, then unitizer will look for a directory named "_pre" in the directory containing the first test file and will treat it as if you had specified it in pre. Any objects created by those scripts will be put into a parent environment for all tests. This provides a mechanism for creating objects that are shared across different test files, as well as loading shared packages. Unlike objects created during test evaluation, any objects created here will not be stored in the unitizer so you will have not direct way to check whether these objects changed across unitizer runs. Additionally, typing ls from the review prompt will not list these objects.
post	NULL, or a character vector pointing to files and/or directories. See pre. If NULL will look for a directory named "_post" in the directory containing the first test file. Scripts are run just prior to exiting unitizer. post code will be run in an environment with the environment used to run pre as the parent. This means that any objects created in pre will be available to post, which you can use to your advantage if there are some things you do in pre you wish to undo in post. Keep in mind that unitizer can manage most aspects of global state, so you should not need to use this parameter to unload packages, remove objects, etc. See details.

history — character(1L) path to file to use to store history generated during interactive unitizer session; the default is an empty string, which leads to unitizer using a temporary file, set to NULL to disable history capture.

interactive.mode — logical(1L) whether to run in interactive mode (request user input when needed) or not (error if user input is required, e.g. if all tests do not pass).

force.update — logical(1L) if TRUE will give the option to re-store a unitizer after re-evaluating all the tests even if all tests passed. You can also toggle this option from the unitizer prompt by typing O (capital letter "o"), though force.update=TRUE will force update irrespective of whether you type O at the prompt

auto.accept — character(X) ADVANCED USE ONLY: YOU CAN EASILY DESTROY YOUR unitizer WITH THIS; whether to auto-accept tests without prompting, use values in c("new", "failed", "deleted", "error") to specify which type(s) of test you wish to auto accept (i.e. same as typing "Y" at the unitizer prompt) or empty character vector to turn off (default)

use.diff — TRUE or FALSE, whether to use diffs when there is an error, if FALSE uses all.equal instead.

show.progress — TRUE or FALSE or integer(1L) in 0:3, whether to show progress updates for each part of the process (TRUE or > 0), for each file processed (TRUE or > 1), and for each test processed (TRUE or > 2).

transcript — TRUE (default in non-interactive mode) or FALSE (default in interactive mode) causes immediate output of stdout/stderr during test evaluation instead of deferred display during test review. This also causes progress updates to display on new lines instead of overlaying on the same line. One limitation of running in this mode is that stderr is no longer captured at all so is unavailable in the review stage. stderr text that is also part of a signalled condition (e.g. "boom" in ‘stop("boom")‘) is still shown with the conditions in the review step. To see direct stderr output in transcript mode scroll up to the test evaluation point.

test.dir — the directory to run the tests on; if NULL will use the “tests/unitizer” package folder if it exists, or “.” if it does not. See infer_unitizer_location) for details.

store.ids — one of

- a function that converts test file names to unitizer ids; if unitizeing multiple files will be lapplyed over each file
- a character vector with unitizer ids, must be the same length as the number of test files being reviewed (see store.id)
- a list of unitizer ids, must be the same length as the number of test files being reviewed; useful when you implement special storage mechanisms for the unitizers (see get_unitizer)

pattern — a regular expression used to match what subset of files in test.dir to unitize

Details

unitize creates unit tests from a single R file, and unitize_dir creates tests from all the R files in the specified directory (analogous to testthat::test_dir).

unitizer stores are identified by unitizer ids, which by default are character strings containing the location of the folder the unitizer RDS files are kept in. unitize and friends will create a unitizer id for you based on the test file name and location, but you can specify your own location as an id, or even use a completely different mechanism to store the unitizer data by implementing S3 methods for get_unitizer and set_unitizer. For more details about storage see those functions.

review allows you to review existing unitizers and modify them by dropping tests from them. Tests are not evaluated in this mode; you are just allowed to review the results of previous evaluations of the tests Because of this, no effort is made to create reproducible state in the browsing environments, unlike with unitize or unitize_dir (see state parameter).

You are strongly encouraged to read through the vignettes for details and examples (browseVignettes("unitizer")). The demo (demo("unitizer")) is also a good introduction to these functions.

Value

unitize and company are intended to be used primarily for the interactive environment and side effects. The functions do return summary data about test outcomes and user input as unitizer_result objects, or for unitize_dir as unitizer_results objects, invisibly. See unitizer_result.

Note

unitizer approximates the semantics of sourcing an R file when running tests, and those of the interactive prompt when reviewing them. The semantics are not identical, and in some cases you may notice differences. For example, when running tests:

- All expressions are run with options(warn=1), irrespective of what the user sets that option to.
- on.exit(...) expressions will be evaluated immediately for top-level statements (either in the test file or in an unitizer_sect, thereby defeating their purpose).
- Each test expression is run in its own environment, which is enclosed by that of previous tests.
- Output and Message streams are sunk so any attempt to debug directly will be near-impossible as you won't see anything.
- For portable tests it is best to use ASCII only string literals (avoiding even escaped bytes or Unicode characters), round numbers, etc., because unitizer uses deparsed test expressions as indices to retrieve reference values. See vignette('u1_intro', package='unitizer') for details and work-arounds.

When reviewing them:

- ls() and q() are over-ridden by unitizer utility functions.
- Expressions are evaluated with options(warn=1) or greater, although unlike in test running it is possible to set and keep options(warn=2).
- Some single upper case letters will be interpreted as unitizer meta-commands.

For a more complete discussion of these differences see the introductory vignette (vignette('u1_intro')), the "Special Semantics" section of the tests vignette (vignette('u2_tests')), and the "Evaluating Expressions at the unitizer Prompt" section of the interactive environment vignette (vignette('u3_interactive-env')).

Default Settings

Many of the default settings are specified in the form `getOption("...")` to allow the user to "permanently" set them to their preferred modes by setting options in their `.Rprofile` file.

See Also

`unitizerState`, `unitizer.opts`, `get_unitizer`, `infer_unitizer_location`, `unitizer_result`

`unitizer` *unitizer*

Description

Simplifies regression tests by comparing objects produced by test code with earlier versions of those same objects. If objects are unchanged the tests pass. 'unitizer' provides an interactive interface to review failing tests or new tests. See vignettes for details.

`unitizer.opts` *Unitizer Options*

Description

Description of major `unitizer` option settings. Once `unitizer` is loaded, you can see a full list of `unitizer` options with `grep("^unitizer", options(), value=TRUE)`.

Basic State Options

Basic state options:

- `unitizer.state`: default state tracking setting (see `unitizerState`)
- `unitizer.seed`: default seed to use when random seed tracking is enabled; this is of type "Wichman-Hill" because it is a lot more compact than the default R random seed, and should be adequate for most unit testing purposes.

Options State Options

Additionally, when tracking option state we set options to what you would find in a freshly loaded vanilla R session, except for systems specific options which we leave unchanged (e.g. `getOption("papersize")`). If you want to add default option values or options to leave unchanged, you can use:

- `unitizer.opts.init`: named list, where names are options, and the associated value is the value to use as the default value for that option when a `unitizer` is launched with options tracking enabled.
- `unitizer.opts.asis`: character, containing regular expressions to match options to leave unchanged (e.g `"^unitizer\."`)

Search Path and Namespace State Options

We also provide options to limit what elements can be removed from the search path and/or have their namespaces unloaded when `unitizer` tracks the search path state. For example, we use this mechanism to prevent removal of the `unitizer` package itself as well as the default R vanilla session packages.

- `unitizer.namespace.keep`: character, names of namespaces to keep loaded (e.g. `"utils"`); note that any imported namespaces imported by namespaces listed here will also remain loaded
- `unitizer.search.path.keep`: character, names of objects to keep on search path (e.g. `"package:utils"`, note the `"package:"`); associated namespaces will also be kept loaded

IMPORTANT: There is a dependency between options tracking and search path / namespace exceptions that stems from most packages setting their default options when they are loaded. As a result, if you add any packages or namespaces to these options and options state tracking is enabled, then you must also add their options to `unitizer.opts.init` or `unitizer.opts.asis` to ensure those options remain loaded or at least set to reasonable values. If you do not do this the packages risk having their options unset.

Some packages cannot be easily loaded and unloaded. For example `data.table` (<= 1.9.5) cannot be unloaded without causing a segfault (see issue #990). For this reason `data.table` is included in `getOption("unitizer.namespace.keep")` by default.

Sytem Default State Options

The following options hold the default system values for the search path / namespace and options state tracking options:

- `unitizer.namespace.keep.base`: namespaces that are known to cause problems when unloaded (as of this writing includes `data.table`)
- `unitizer.search.path.keep.base`: vanilla R session packages, plus `"package:unitizer"` and `"tools:rstudio"`, the latter because its implementation prevents re-attaching it if it is detached.
- `unitizer.opts.asis.base`: system specific options that should not affect test evaluation (e.g. `getOption("editor")`).
- `unitizer.opts.init.base`: base options (e.g. `getOption("width")` that will be set to what we believe are the factory settings for them.

These are kept separate from the user specified ones to limit the possibility of inadvertent modification. They are exposed as options to allow the user to unset single values if required, though this is intended to be rare. `unitizer` runs with the union of user options and the system versions described here. For `unitizer.opts.init`, any options set that are also present in `unitizer.opts.init.base` will overrule the base version.

Display / Text Capture Options

These options control how `unitizer` displays data such as diffs, test results, etc.

- unitizer.test.out.lines: integer(2L), where first values is maximum number of lines of screen output to show for each test, and second value is the number of lines to show if there are more lines than allowed by the first value
- unitizer.test.msg.lines: like unitizer.test.out.lines, but for stderr output
- unitizer.test.fail.context.lines: integer(2L), used exclusively when comparing new to references tests when test faile; first values is maximum number of lines of context to show around a test, centered on differences if there are any, and second value is the number of context lines to show if using the first value is not sufficient to fully display the test results
- unitizer.show.output: TRUE or FALSE, whether to display test stdout and stderr output as it is evaluated.
- unitizer.disable.capt: logical(2L), not NA, with names c("output", "message") where each value indicates whether the corresponding stream should be captured or not. For stdout the stream is still captured but setting the value to FALSE tees it.
- unitizer.max.capture.chars: integer(1L) maximum number of characters to allow capture of per test
- unitizer.color whether to use ANSI color escape sequences, set to TRUE to force, FALSE to force off, or NULL to attempt to auto detect (based on code from package:crayon, thanks Gabor Csardi)
- unitizer.use.diff TRUE or FALSE, whether to use a diff of test errors (defaults to TRUE)

Misc Options

- unitizer.history.file character(1L) location of file to use to store history of command entered by user in in interactive unitizer prompt; "" is interpreted as tempfile()
- unitizer.prompt.b4.quit.time integer(1L) unitizers that take more seconds than this to evaluate will post a confirmation prompt before quitting; this is to avoid accidentally quitting after running a unitizer with many slow running tests and having to re-run them again.

See Also

unitizerState

---

unitizerList *S4 Object To Implement Base List Methods*

Description

Internal unitizer objects used to manage lists of objects. The only user facing instance of these objects are conditionList objects. For the most part these objects behave like normal S3 lists. The list contents are kept in the .items slot, and the following methods are implemented to make the object mostly behave like a standard R list: [, [[, [<-, [[<-, as.list, append, length, names, and names<-.

Details

The underlying assumption is that the ‘.items‘ slot is a list (or an expression), and that slot is the only slot for which it’s order and length are meaningful (i.e. there is no other list or vector of same length as ‘.items‘ in a different slot that is supposed to map to ‘.items‘). This last assumption allows us to implement the subsetting operators in a meaningful manner.

The validity method will run `validObject` on the first, last, and middle items (if an even number of items, then the middle closer to the first) assuming they are S4 objects. We don’t run on every object to avoid potentially expensive computation on all objects.

Slots

`.items` a list or expression

`.pointer` integer, used for implementing iterators

`.seek.fwd` logical used to track what direction iterators are going

See Also

`conditionList`

Examples

```
new('unitizerList', .items=list(1, 2, 3))
```

---

unitizerState *Tests and Session State*

Description

While R generally adheres to a "functional" programming style, there are several aspects of session state that can affect the results of code evaluation (e.g. global environment, search path). `unitizer` provides functionality to increase test reproducibility by controlling session state so that it is the same every time a test is run. This functionality is turned off by default to comply with CRAN requirements, and also because there are inherent limitations in R that may prevent it from fully working in some circumstances. You can permanently enable the suggested state tracking level by adding `options(unitizer.state='suggested')` in your `.Rprofile`, although if you intend to do this be sure to read the “CRAN non-compliance” section.

Usage

```
state(
  par.env,
  search.path,
  options,
  working.directory,
  random.seed,
  namespaces
```

```
)

in_pkg(package = NULL)
```

Arguments

par.env	NULL to use the special unitizer parent environment, or an environment to use as the parent environment, or the name of a package as a character string to use that packages' namespace as the parent environment, or a unitizerInPkg object as produced by in_pkg, assumes .GlobalEnv if unspecified
search.path	one of 0:2, uses the default value corresponding to getOption(unitizer.state), which is 0 in the default unitizer state of "off". See "Custom Control" section for details.
options	same as search.path
working.directory	same as search.path
random.seed	same as search.path
namespaces	same as search.path
package	character(1L) or NULL; if NULL will tell unitize to attempt to identify if the test file is inside an R package folder structure and if so run tests in that package's namespace. This should work with R CMD check tests as well as in normal usage. If character will take the value to be the name of the package to use the namespace of as the parent environment. Note that in_pkg does not retrieve the environment, it just tells unitize to do so.

Value

for state a unitizerStateRaw object, for in_pkg a unitizerInPkg object, both of which are suitable as values for the state parameter for unitize or as values for the "unitizer.state" global option.

CRAN Non-Compliance and Other Caveats

In the default state management mode, this package fully complies with CRAN policies. In order to implement advanced state management features we must lightly trace some base functions to alert unitizer each time the search path is changed by a test expression. The traced function behavior is completely unchanged other than for the side effect of notifying unitizer each time they are called. Additionally, the functions are only traced during unitize evaluation and are untraced on exit. Unfortunately this tracing is against CRAN policies, which is why it is disabled by default.

Arguably other aspects of state management employed outside of state="default" _could_ be considered CRAN non-compliant, but none of these are deployed unless you explicitly chose to do so. Additionally, unitizer limits state manipulation to the evaluation of its processes and restores state on exit. Some exceptional failures may prevent restoring state fully.

If state management were to fail fail in an unhandled form, the simplest work-around is to turn off state management altogether with state="default". If it is a particular aspect of state management (e.g. search paths with packages attached with devtools::load_all), you can disable just that aspect of state (see "Custom Control" section).

For more details see the reproducible tests vignette with: vignette(package='unitizer', 'u4_reproducible-tests')

Overview

You can control how unitizer manages state via the state argument to unitize or by setting the "unitizer.state" option. This help file discusses state management with unitizer, and also documents two functions that, in conjunction with unitize or unitize_dir allow you to control state management.

Note: most of what is written in this page about unitize applies equally to unitize_dir.

unitizer provides functionality to insulate test code from variability in the following. Note the "can be" wording because by default these elements of state are not managed:

- Workspace / Parent Environment: all tests can be evaluated in environments that are children of a special environment that does not inherit from .GlobalEnv. This prevents objects that are lying around in your workspace from interfering with your tests.
- Random Seed: can be set to a specific value at the beginning of each test file so that tests using random values get the same value at every test iteration. This only sets the seed at the beginning of each test file, so changes in order or number of functions that generate random numbers in your test file will affect subsequent tests. The advantage of doing this over just setting the seed directly in the test files is that unitizer tracks the value of the seed and will tell you the seed changed for any given test (e.g. because you added a test in the middle of the file that uses the random seed).
- Working Directory: can be set to the tests directory inside the package directory if the test files appear to be inside the folder structure of a package, and the test file does not appear to be run as part of a check run (e.g. R CMD check, 'tools::testInstalledPakage'). If test files are not inside a package directory structure then can be set to the test files' directory.
- Search Path: can be set to what you would typically find in a freshly loaded vanilla R session. This means any non default packages that are loaded when you run your tests are unloaded prior to running your tests. If you want to use the same libraries across multiple tests you can load them with the pre argument to unitize or unitize_dir. Due to limitations of R this is only an approximation to actually restarting R into a fresh session.
- Options: same as search path, but see "Namespaces" next.
- Namespaces: same as search path; this option is only made available to support options since many namespaces set options onLoad, and as such it is necessary to unload and re-load them to ensure default options are set. See the "Namespaces and Options" section.

In the "suggested" state tracking mode (previously known as "recommended"), parent environment, random seed, working directory, and search path are all managed to level 2, which approximates what you would find in a fresh session (see "Custom Control" section below). For example, with the search path managed, each test file will start evaluation with the search path set to the tests folder of your package. All these settings are returned to their original values when unitizer exits.

To manage the search path unitizer detaches and re-attaches packages. This is not always the same as loading a package into a fresh R session as detaching a package does not necessarily undo every action that a package takes when it is loaded. See detach for potential pitfalls of enabling this setting. Additionally, packages attached in non-standard ways (e.g. devtools::load_all) may not re-attach properly.

You can modify what aspects of state are managed by using the state parameter to unitize. If you are satisfied with basic default settings you can just use the presets described in the next section.

If you want more control you can use the return values of the state and in_pkg functions as the values for the state parameter for unitize.

State is reset after running each test file when running multiple test files with unitize_dir, which means state changes in one test file will not affect the next one.

State Presets

For convenience unitizer provides several state management presets that you can specify via the state parameter to unitize. The simplest method is to specify the preset name as a character value:

- "suggested":
  - Use special (non .GlobalEnv) parent environemnt
  - Manage search path
  - Manage random seed (and set it to be of type "Wichmann-Hill" for space considerations).
  - Manage workign directory
  - Leave namespace and options untouched
- "safe" like suggested, but turns off tracking for search path in addition to namespaces and options. These settings, particularly the last two, are the most likely to cause compatibility problems.
- "pristine" implements the highest level of state tracking and control
- "basic" keeps all tracking, but at a less aggressive level; state is reset between each test file to the state before you started unitizeing so that no single test file affects another, but the state of your workspace, search path, etc. when you launch unitizer will affect all the tests (see the Custom Control) section.
- "off" (default) state tracking is turned off

Custom Control

If you want to customize each aspect of state control you can pass a unitizerState object as the state argument. The simplest way to do this is by using the state constructor function. Look at the examples for how to do this.

For convenience unitize allows you to directly specify a parent environment if all you want to change is the parent evaluation environment but are otherwise satisfied with the defaults. You can even use the in_pkg function to tell unitizer to use the namespace associated with your current project, assuming it is an R package. See examples for details.

If you do chose to modify specific aspects of state control here is a guide to what the various parameter values for state do:

- For par.env: any of the following:
  - NULL to use the special unitizer parent environment as the parent environment; this environment has for parent the parent of .GlobalEnv, so any tests evaluated therein will not be affected by objects in .GlobalEnv see (vignette("unitizer_reproducible_state")).
  - an environment to use as the parent evaluation environment
  - the name of a package to use that package's namespace environment as the parent environment

  - the return value of `in_pkg`; used primarily to autodetect what package namespace to use based on package directory structure

- For all other slots, the settings are in `0:2` and mean:
  - 0 turn off state tracking
  - 1 track, but start with state as it was when `unitize` was called.
  - 2 track and set state to what you would typically find in a clean R session, with the exception of `random.seed`, which is set to `getOption("unitizer.seed")` (of kind "Wichmann-Hill" as that seed is substantially smaller than the R default seed).

If you chose to use level `1` for the random seed you should consider picking a random seed type before you start unitizer that is small like "Wichman-Hill" as the seed will be recorded each time it changes.

Permanently Setting State Tracking

You can permanently change the default state by setting the "unitizer.state" option to the name of the state presets above or to a or to a state settings option object generated with `state` as described in the previous section.

Avoiding `.GlobalEnv`

For the most part avoiding `.GlobalEnv` leads to more robust and reproducible tests since the tests are not influenced by objects in the workspace that may well be changing from test to test. There are some potential issues when dealing with functions that expect `.GlobalEnv` to be on the search path. For example, `setClass` uses `topenv` to find a default environment to assign S4 classes to. Typically this will be the package environment, or `.GlobalEnv`. However, when you are in `unitizer` this becomes the next environment on the search path, which is typically locked, which will cause `setClass` to fail. For those types of functions you should specify them with an environment directly, e.g. `setClass("test", slots=c(a="integer"), where=environment())`.

Namespaces and Options

Options and namespace state management require the ability to fully unload any non-default packages and namespaces, and there are some packages that cannot be unloaded, or should not be unloaded (e.g. data.table). I some systems it may even be impossible to fully unload any compiled code packages (see `detach`. If you know the packages you typically load in your sessions can be unloaded, you can turn this functionality on by setting `options(unitizer.state="pristine")` either in your session, in your `.Rprofile` file, or using `state="prisitine"` in each call to `unitize` or `unitize_dir`. If you have packages that cannot be unloaded, but you still want to enable these features, see the "Search Path and Namespace State Options" section of `unitizer.opts` docs.

If you run `unitizer` with options and namespace tracking and you run into a namespace that cannot be unloaded, or should not be unloaded because it is listed in `getOption("unitizer.namespace.keep")`, `unitizer` will turn off `options` state tracking from that point onwards.

Additionally, note that `warn` and `error` options are always set to `1` and `NULL` respectively during test evaluation, irrespective of what option state tracking level you select.

Known Untracked State Elements

- system time: tests involving functions such as `date` will inevitably fail
- locale: is not tracked because it so specific to the system and so unlikely be be changed by user action; if you have tests that depend on locale be sure to set the locale via the `pre` argument to `unitize`, and also to reset it to the original value in `post`.

See Also

`unitize`, `unitizer.opts`

Examples

```
## Not run:
## In this examples we use `...` to denote other arguments to `unitize` that
## you should specify.  All examples here apply equally to `unitize_dir`

## Run with suggested state tracking settings
unitize(..., state="suggested")
## Manage as much of state as possible
unitize(..., state="pristine")

## No state management, but evaluate with custom env as parent env
my.env <- new.env()
unitize(..., state=my.env)
## use custom environment, and turn on search.path tracking
## here we must use the `state` function to construct a state object
unitize(..., state=state(par.env=my.env, search.path=2))

## Specify a namespace to run in by name
unitize(..., state="stats")
unitize(..., state=state(par.env="stats")) # equivalent to previous

## Let `unitizer` figure out the namespace from the test file location;
## assumes test file is inside package folder structure
unitize("mytests.R", state=in_pkg()) # assuming mytests.R is part of a pkg
unitize("mytests.R", state=in_pkg("mypkg")) # also works

## End(Not run)
```

---

unitizer_demo *Demo Details and Helper Functions*

Description

`unitizer` provides an interactive demo you can run with `demo("unitizer")`.

Usage

```
`[Press ENTER to Continue]`()

show_file(f, width = getOption("width", 80L))

copy_fastlm_to_tmpdir()

update_fastlm(dir, version)

unitizer_check_demo_state()

unitizer_cleanup_demo()
```

Arguments

f	path to a file
width	display width in characters
dir	path to the temporary package
version	one of "0.1.0", "0.1.1", "0.1.2"

Value

character(1L)

Demo Details

The demo centers around simulated development of the utzflm package. unitizer includes in its sources three copies of the source code for the utzflm package, each at a different stage of development. This allows us to create reference unitizer tests under one version, move to a new version and check for regressions, and finally fix the regressions with the last version. The version switching is intended to represent the package development process.

The demo manages the utzflm code changes, but between each update allows the user to interact with unitizer. The demo operates under the assumption that the user will accept the first set of tests and reject the failing tests after the first update. If the user does anything different then the demo commentary may not apply anymore.

utzflm

utzflm is a "dummy" package that implements a faster computation of slope, intercept, and R^2 for single variable linear regressions than is available via summary(lm()...).

Helper Functions

copy_fastlm_to_tmpdir copies the initial version of the utzflm sources to a temporary directory, show_file displays the contents of a source code file, update_fastlm changes the source code of utzflm, and unitizer_check_demo_state and unitizer_cleanup_demo perform janitorial functions. None of these functions are intended for use outside of the unitizer demo.

unitizer_result *Return Values and Related Methods for* unitize *Functions*

Description

unitize and related functions are run primarily for the interactive environment they provide and for their side effects (updating stored unitizer objects), but the return values may be useful under some circumstances if you need to retrieve test status, user selections, etc..

Usage

```
## S3 method for class 'unitizer_result'
print(x, ...)

## S3 method for class 'unitizer_results'
print(x, ...)
```

Arguments

x	the object to print
...	extra arguments for print generic

Details

unitize and review return a unitizer_result S3 object. This is a data frame that contains details about the status of each test. unitize_dir returns a unitize_results S3 object, which is a list of unitize_result objects.

Both unitize_results and unitize_result have print methods documented here. In addition to the print methods, both of the result objects have get_unitizer methods so that you can retrieve the stored unitizer objects.

Please note that with unitize_dir you can re-review a single unitizer several times during during a single call to unitize_dir. This is to allow you to re-evaluate specific unitizers easily without having to re-run the entire directory again. Unfortunately, as a result of this feature, the return values of unitize_dir can be misleading because they only report the result of the last review cycle.

Additionally, unitize_dir will report user selections during the last review even if in the end the user chose not to save the modified unitizer. You will be alerted to this by an onscreen message from the print method (this is tracked in the "updated" attribute of the unitizer_result object). Finally, if in the last iteration before exit you did not save the unitizer, but you did save it in previous review cycles in the same unitize_dir call, the displayed selections and test outcomes will correspond to the last unsaved iteration, not the one that was saved. You will be alerted to this by an on-screen message (this is tracked through the "updated.at.least.once" attribute of the unitizer_result object).

Value

x, invisibly

See Also

unitize, get_unitizer

`unitizer_sect` *Define a* `unitizer` *Section*

Description

The purpose of `unitizer` sections is to allow the user to tag a group of test expressions with meta information as well as to modify how tests are determined to pass or fail.

Usage

```
unitizer_sect(
  title = NULL,
  expr = expression(),
  details = character(),
  compare = new("testFuns")
)
```

Arguments

`title`	character 1 length title for the section, can be omitted though if you do omit it you will have to refer to the subsequent arguments by name (i.e. `unitizer_sect(expr=...)`)
`expr`	test expression(s), most commonly a call to `{}` with several calls inside (see examples)
`details`	character more detailed description of what the purpose of the section is; currently this doesn't do anything.
`compare`	a function or a `testFuns` object

Tested Data

`unitizer` tracks the following:

- value: the return value of the test
- conditions: any conditions emitted by the test (e.g. warnings or errors)
- output: screen output
- message: stderr output
- aborted: whether the test issued an 'abort' restart (e.g. by calling 'stop' directly or indirectly)

In the future stdout produced by the test expression itself may be captured separately from that produced by print/showing of the return value, but at this point the two are combined.

Each of the components of the test data can be tested, although by default only `value` and `condition` are checked. Testing `output` is potentially duplicative of testing `value`, since most often `value` is printed to screen and the screen output of the value closely correlates to the actual value. In some cases it is useful to explicitly test the `output`, such as when testing `print` or `show` methods.

Comparison Functions

The comparison function should accept at least two parameters, and require no more than two. For each test component, the comparison function will be passed the reference data as the first argument, and the newly evaluated data as the second. The function should return TRUE if the compared test components are considered equivalent, or FALSE. Instead of FALSE, the function may also return a character vector describing the mismatch, as `all.equal` does.

WARNING: Comparison functions that set and/or unset `sink` can potentially cause problems. If for whatever reason you must really sink and unsink output streams, please take extreme care to restore the streams to the state they were in when the comparison function was called.

Any output to `stdout` or `stderr` is captured and only checked at the end of the `unitizer` process with the expectation that there will be no such output.

`value` and `conditions` are compared with `all_eq`, which is a wrapper to `all.equal` except that it returns FALSE instead of a descriptive string on failure. This is because `unitizer` will run `diffObj` on the test data components that do not match and including the `all.equal` output would be redundant.

If a comparison function signals a condition (e.g. throws a warning) the test will not be evaluated, so make sure that your function does not signal conditions unless it is genuinely failing.

If you wish to provide custom comparison functions you may do so by passing an appropriately initialized `testFuns` object as the value to the `compare` parameter to `unitizer_sect` (see examples).

Make sure your comparison functions are available to `unitize`. Comparisons will be evaluated in the environment of the test. By default `unitize` runs tests in environments that are not children to the global environment, so functions defined there will not be automatically available. You can either specify the function in the test file before the section that uses it, or change the base environment tests are evaluated in with `unitize(..., par.env)`, or make sure that the package that contains your function is loaded within the test script.

Nested Sections

It is possible to have nested sections, but titles, etc. are ignored. The only effect of nested sections is to allow you to change the comparison functions for a portion of the outermost `unitizer_sect`.

Note

if you want to modify the functions used to compare conditions, keep in mind that the conditions are stored in `conditionList` objects so your function must loop through the lists and compare conditions pairwise. By default `unitizer` uses the `all.equal` method for S4 class `conditionList`.

`untizer` does not account for sections when matching new and reference tests. All tests will be displayed as per the section they belong to in the newest version of the test file, irrespective of what section they were in when the tests were last run.

Calls to `unitizer_sect` should be at the top level of your test script, or nested within other `unitizer_sects` (see "Nested Sections"). Do not expect code like `(untizer_sect(..., ...))` or `{unitizer_sect(..., ...)}` or `fun(unitizer_sect(..., ...))` to work.

See Also

`testFuns`, `all_eq`

Examples

```
unitizer_sect("Switch to `all.equal` instead of `all_eq`",
  {
    fun(6L)
    fun("hello")
  },
  compare=testFuns(value=all.equal, conditions=all.equal)
)
unitizer_sect("Use identical for ALL test data, including stdout, etc.",
  {
    fun(6L)
    fun("hello")
  },
  compare=identical
)
```

$.unitizerItem *Retrieve Test Contents From Test Item*

Description

Intended for use within the unitizer interactive environment, allows user to retrieve whatever portions of tests are stored by unitizer.

Usage

```
## S4 method for signature 'unitizerItem'
x$name

## S4 method for signature 'unitizerItem,ANY'
x[[i, j, ..., exact = TRUE]]
```

Arguments

x	a unitizerItem object, typically .NEW or .REF at the unitizer interactive prompt
name	a valid test sub-component
i	a valid test sub-component as a character string, or a sub-component index
j	missing for compatibility with generic
...	missing for compatibility with generic
exact	unused, always matches exact

Details

Currently the following elements are available:

- `call` the call that was tested as an unevaluated call, but keep in mind that if you intend to evaluate this for a reference item the environment may not be the same so you could get different results (`ls` will provide more details)
- `value` the value that results from evaluating the test, note this is equivalent to using `.new` or `.ref`; note that the value is displayed using `desc` when viewing all of `.NEW` or `.REF`
- `output` the screen output (i.e. anything produced by cat/print, or any visible evaluation output) as a character vector
- `message` anything that was output to `stderr`, mostly this is all contained in the conditions as well, though there could be other output here, as a character vector
- `conditions` a `conditionList` containing all the conditions produced during test evaluation
- `aborted` whether the test call issues a restart call to the 'abort' restart, as 'stop' does.

Value

the test component requested

Examples

```
## From the unitizer> prompt:
.NEW <- mock_item()  # .NEW is normally available at unitizer prompt
.NEW$call
.NEW$conditions
.NEW$value              # equivalent to `.new`
```

Index