archimedes Documentation

Release 0.3.0

Tuukka Turto

February 17, 2017

Contents:

CHAPTER 1

Intro

Archimedes is a collection of macros for writing tests. It's geared towards nose, but other test runners that work in similar manner should work too.

Main goal for Archimedes is to make testing fun and as easy as possible.

Also, Archimedes was a Greek mathematician, physicist, engineer, inventor and astronomer.

Installation

Preferred installation method is pip (in virtualenv or equivalent):

```
pip install libarchimedes
```

Facts and checks

Basic building block is of course a test case. Archimedes follows nose convention, where test function name starts with "**test_**", so they're easy to collect and execute programmatically. To define a test case, `fact` macro is used:

```
(fact "this is a test case"
  (assert (= 1 1)))
```

This will define a function, which is equivalent to:

```
(defn test_this_is_a_test_case []
  "this is a test case"
  (assert (= 1 1)))
```

Nose (or any other test framework that follows the convention) can then programmatically find this and execute it.

Sometimes one might want to execute test case immediately. This can be useful when working in interactive mode, be it Hy repl or Jupyter notebook. For these situations, there is `check` macro. It defines test case just like `fact` macro does and then executes it:

```
(check "this is executed immediately"
  (assert (= 1 1)))
```

Both of these support specifying common setup code that can be shared between several test cases. `background` macro specifies setup code with a unique name and `with-background` takes one or more variables from that specification in use. Since this probably sounds a bit confusing, an example is in order:

```
(background some-numbers
  a 3
  b 4
  c 5)

(fact "sum of two numbers"
  (with-background some-numbers [a b]
    (assert (= (+ a b) 7))))

(fact "product of three numbers"
  (with-background some-numbers [a b c]
    (assert (= (* a b c) 60))))
```

Background can contain arbitrarily many variable definitions and they can be more complex than simple values (calculations for example).

Hypothesis

Hypothesis is a Python library for property based testing, similar to what QuickCheck in Haskell. Archimedes provides few parameters for `fact` and `check` macros that are used to instruct Hypothesis to generate test data.

Note: Some knowledge of Hypothesis is assumed for this section.

Three macros are provided for controlling Hypothesis: `variants`, `sample` and `profile`.

`variants` macro controls test data generation. It maps into `given` decorator in Hypothesis. Body of `variants` consists of two or more items. Every odd specifies variable name and element after that is strategy specifying what kind of data to generate.

`sample` maps to `example` decorator. It specifies concrete examples for variables to check. Other than that, it works just like `variants` macro (variable, value).

`profile` maps into `settings` decorator in Hypothesis. It is used to tweak behaviour of Hypothesis for a specific test case.

Below is an example test case that showcases usage of all these elements.

```
(require [archimedes [fact]])
(import [hypothesis.strategies [integers]])

(fact "sum of two positive numbers is larger than either one of them"
   (variants :a (integers :min-size 1)
             :b (integers :min-size 1))
   (sample :a 0 :b 0)
   (profile :max-examples 500)
   (assert (> (+ a b) a))
   (assert (> (+ a b) b)))
```

This causes test case to be run at maximum of 500 times. There are two parameters `a` and `b`, which both are integers and have value of 1 or greater. There is also a specific test case for them being zero.

CHAPTER 4

Errors

Sometimes it's useful to verify that a certain exception is raised. This is achieved with `assert-error` or `assert-macro-error` macro. Both take two parameters: a string and piece of code. The code is executed and resulting exception is compared with the provided string. In case of `assert-error` this comparison is done by simply calling `str` for exception. For `assert-macro-error` message attribute is used. If no exception is raised, or raised exception doesn't match the provided string, assertion fails.

```
(fact "errors can be asserted"
      (assert-error "error"
                    (raise (ValueError "error"))))
```

```
(fact "macro errors can be asserted"
      (assert-macro-error "cond branches need to be a list"
                          (cond (= 1 1) true)))
```

CHAPTER 5

Hamcrest

Note: Hamcrest is't installed by default

hamcrest is a framework for matcher objects. It's very useful for writing expressive tests that are easy to read and report failures in user friendly way. While hamcrest comes with pleothra of predefined matchers, part of the power is extensibility of the framework. Developer can write custom matchers for their specific problem domain.

There are two macros for defining matchers: `defmatcher` and `attribute-matcher`.

`defmatcher` is used to define a generic matcher. Simple example below shows how this is done:

```
(defmatcher is-zero? []
            :match? (= item 0)
            :match! "a zero"
            :no-match! (.format "was a value of {0}" item))

(assert-that 0 (is-zero?))
```

Parameters are: name of the matcher, input parameters, rule for matching, how to report match and how to report mismatch. There is special variable `item` that corresponds to the item that the matcher is being run against.

In case where more general matcher is wanted, an input parameters can be used. These will be available as properties of `self` in methods:

```
(defmatcher in-between? [a b]
            :match? (< a item b)
            :match! (.format "value between {0} and {1}" self.a self.b)
            :no-match! (.format "was a value of {0}" item))

(assert-that 5 (in-between? 2 7))
```

While `defmatcher` is powerful and can be used to define all kinds of matchers, sometimes a simple one is needed. For these times `attribute-matcher` is sufficient:

```
(attribute-matcher item-with-length?
                   len =
                   "an item with length {0}"
                   "was an item with length {0}")

(assert-that "foo" (is- (item-with-length? 3)))
```

Second parameter defines a function that is run for item being matched and third parameter defines what kind of comparison should be performed. This macro will always result to a matcher that takes a single parameter defining some sort of value, which is then compared to result of the function. Handy trick is to use dot notation to access object method:

```
(attribute-matcher object-with-name?
                   .name =
                   "an object with name {0}"
                   "was an object with name {0}")

(assert-that obj (is- (object-with-name? "foo")))
```

Hymn

Note: hymn is not installed by default

hymn is a monad libary. While it's perfectly possible to test code that uses monads without Archimedes, there is some patterns that keep repeating. Those patterns (or rather one of them) has been collected in Archimedes.

In cases where code being tested produces `Either` there are generally several different considerations to be made: is it `left` or `right`? Does it have correct value inside of it. `assert-right` captures pattern of the first case. It first checks that code being executed produced `right` and then proceeds to perform additional checks. If any of these fail, the assert fails too.

```
(assert-right (do-monad [status (advance-time-m society)]
                        status)
              (assert-that society
                           (has-less-resources-than? old-resources)))
```

CHAPTER 7

Indices and tables

- genindex
- search