pyqcy Documentation

Release 0.4.4

Karol Kuczmarski

September 26, 2015

Contents

pyqcy [pyksi:] is a test framework that supports unique testing model, inspired by the brilliant *QuickCheck* library for Haskell. Rather than writing fully-fledged test cases, you only need to define logical **properties** that your code has to satisfy. Based on that, *pyqcy* will automatically generate test cases for you - hundreds of them, in fact!

CHAPTER 1

Example

```
from pyqcy import qc, int_, main

@qc
def addition_actually_works(
    x=int_(min=0), y=int_(min=0)
):
    the_sum = x + y
    assert the_sum >= x and the_sum >= y

if __name__ == '__main__':
    main()
```

```
$ python ./example.py
addition_actually_works: passed 100 tests.
```

Yes, that's 100 distinct test cases. *pyqcy* has generated them all for you!

CHAPTER 2

Installation

Either from PyPI:

```
$ pip install pyqcy
```

or directly from GitHub if you want the bleeding edge version:

```
$ git clone git://github.com/Xion/pyqcy.git
$ cd pyqcy
$ ./setup.py develop
```

CHAPTER 3

Learn more

3.1 Defining properties

In *pyqcy*, the test properties are defined as regular Python functions but they are all adorned with the `qc()` decorator.

Here's an example:

```
from pyqcy import *

@qc
def sorting_preserves_length(
    l=list_(of=int, min_length=1, max_length=128)
):
    before_sort = l
    after_sort = list(sorted(l))
    assert len(before_set) == len(after_sort)
```

Inside the function, we use its parameters as a sort of **quantified variables**. As you can see, their defaults are somewhat unusual: they specify how to obtain *arbitrary* (random) values for those variables. *pyqcy* will take those specifications, use them to automatically generate test data and then invoke your property's code.

Note: For more information about different way of running tests for your properties, check the documentation on that.

`pyqcy.qc([tests])`

Decorator for Python functions that define properties to be tested by pyqcy.

It is expected that default values for function arguments define generators that will be used to generate data for test cases. See the section about using generators for more information.

Example of using `@qc` to define a test property:

```
    @qc
    def len_behaves_correctly(
        l=list_(int, min_length=1, max_length=64)
    ):
        assert len(l) == l.__len__()
```

Parameters **tests** – Number of tests to execute for this property. If omitted, the default number of 100 tests will be executed.

3.2 Using generators

To provide test data for your properties, *pyqcy* has a set of generators for all common types and use cases, including Python's *scalar types* and *collections*. It is also easy to *combine* several generators into one - up to creating *complex data structures* on the fly.

Still, if those are not enough, you can always define your own generator. This is especially handy for custom classes, as it enables you to write properties that should be true for their instances. To create a custom generator, simply define a function that returns an appropriate random object and decorate it with the `arbitrary()` decorator:

```
from pyqcy import *

@arbitrary(MyClass)
def my_class():
    obj = MyClass()
    obj.some_field = next(int_(min=0, max=1024))
    obj.other_field = next(str_(max_length=64))
    return obj
```

Now you can write properties which use the new generator:

```
@qc
def forbs_correctly(obj=MyClass):
    assert obj.forb() >= obj.some_field * len(obj.other_field)
```

Because we have passed a type argument to `arbitrary()`, we can use our class name (`MyClass`) in place of generator name (`my_class`) - although the latter is of course still possible.

`pyqcy.arbitraries.`**`arbitrary`**(*type_=None*)

Decorator to be applied on functions in order to turn them into generators of arbitrary ("random") values of given type.

Parameters **`type`** – Type of values generated by the function

The `type_` argument is optional. If provided, objects returned by the function will be checked against this type. It will be also possible to use the type directly when defining properties.

Examples:

```
    from pyqcy import *

    @arbitrary(MyClass)
    def my_class():
        return MyClass()

    @qc
    def my_class_works(obj=MyClass):
        assert obj.is_valid()
```

3.2.1 Built-in types

Most Python types are conveniently supported by *pyqcy* and generators for them are readily available. They should cover a vast majority of typical use cases.

Numeric types

Numeric types have parametrized generators that allow for setting desired range of produces values. But if we are fine with the defaults, we can simply use the types directly, as seen in this example:

```
@qc
def vec2d_length_is_positive(x=float, y=float):
    return vec2d_len(x, y) >= 0.0
```

pyqcy.arbitraries.numbers.**int_**(*min*, *max*)
Generator for arbitrary integers.

By default, it generates values from the whole integer range supported by operating system; this can be adjusted using parameters.

Parameters

- **min** – A minimum value of integer to generate
- **max** – A maximum value of integer to generate

pyqcy.arbitraries.numbers.**float_**(*min*, *max*)
Generator for arbitrary floats.

Parameters

- **min** – A minimum value of float to generate
- **max** – A maximum value of float to generate

pyqcy.arbitraries.numbers.**complex_**(*min_real*, *max_real*, *min_imag*, *max_imag*)
Generator for arbitrary complex numbers of the built-in Python complex type.

Parameters for this generator allow for adjusting the rectangle on the complex plane where the values will come from.

Parameters

- **min_real** – A minimum value for real part of generated numbers
- **max_real** – A maximum value for real part of generated numbers
- **min_imag** – A minimum value for the imaginary part of generated numbers
- **max_imag** – A maximum value for the imaginary part of generated numbers

Strings

For creating arbitrary texts, *pyqcy* has two generators for ANSI and Unicode strings. You can specify what characters the generators should draw from, as well the minimum and maximum length of strings to generate.

pyqcy.arbitraries.strings.**str_**(*of*, *min_length*, *max_length*)
Generator for arbitrary strings.

Parameters for this generator allow for adjusting the length of resulting strings and the set of characters they are composed of.

Parameters

- **of** – Characters used to construct the strings. This can be either an iterable of characters (e.g. a string) or a generator that produces them.
- **min_length** – A minimum length of string to generate

- **max_length** – A maximum length of string to generate

pyqcy.arbitraries.strings.**unicode_**(*of*, *min_length*, *max_length*)
Generator for arbitrary Unicode strings.

Parameters for this generator allow for adjusting the length of resulting strings and the set of characters they are composed of.

Parameters

- **of** – Characters used to construct the strings. This can be either an iterable of characters (e.g. a string) or a generator that produces them.
- **min_length** – A minimum length of string to generate
- **max_length** – A maximum length of string to generate

Quite often you would also want to deal only with strings of certain form that matches the expected input of the code you are testing. In those cases it's useful to specify a regular expression that autogenerated strings should match.

pyqcy.arbitraries.strings.**regex**(*pattern*)
Generator for strings matching a regular expression.

Parameters **pattern** – A regular expression - either a compiled one (through `re.compile()`) or a string pattern

---

Note: Currently the *regex* reverser supports only a limited subset of syntactic features offered by Python regular expressions. For example, it doesn't support negative matches on character sets (`[^...]`) or backreferences to capture groups (`\\1`, `\\2`, etc.).

Tuples

Tuples can be produced by combining several generators together through *tuple_()* function. There are also handy shortcuts for pairs, triplers and quadruples that consists of values from the same source.

pyqcy.arbitraries.collections.**tuple_**(**generators*, *of*, *n*)
Generator for arbitrary tuples.

The tuples are always of the same length but their values may come from different generators. There two ways to specify those generators - either enumerate them all:

```
tuple_(int_(min=0, max=255), str_(max_length=64))
```

or use `n` argument with a single generator to get uniform tuples:

```
ip_addresses = tuple_(int_(min=0, max=255), n=4)
ip_addresses = tuple_(of=int_(min=0, max=255), n=4)
```

Those two styles are mutually exclusive - only one can be used at a time.

Parameters

- **of** – Generator used to generate tuple values
- **n** – Tuple length

pyqcy.arbitraries.collections.**two**(*of*)
partial(func, *args, **keywords) - new function with partial application of the given arguments and keywords.

pyqcy.arbitraries.collections.**three**(*of*)
partial(func, *args, **keywords) - new function with partial application of the given arguments and keywords.

pyqcy.arbitraries.collections.**four**(*of*)
partial(func, *args, **keywords) - new function with partial application of the given arguments and keywords.

Collections

Lists and dictionaries can be generated by giving their minimum and maximum size, as well as a generator for their elements. For dictionaries, you can either specify a separate generators for keys and values, or a single generator that outputs 2-element tuples.

pyqcy.arbitraries.collections.**list_**(*of*, *min_length*, *max_length*)
Generator for arbitrary lists.

Parameters for this generator allow for adjusting the length of resulting list and elements they contain.

Parameters

- **of** – Generator for list elements
- **min_length** – A minimum length of list to generate
- **max_length** – A maximum length of list to generate

Example of test property that uses *list_()*:

```
@qc
def calculating_average(
    l=list_(of=int_(min=0, max=1024),
            min_length=16, max_length=2048)
):
    average = sum(l) / len(l)
    assert min(l) <= average <= max(l)
```

pyqcy.arbitraries.collections.**dict_**(*keys*, *values*, *items*, *min_length*, *max_length*)
Generator for arbitrary dictionaries.

Dictionaries are specified using generators - either for `keys` and `values` separately:

```
dict_(keys=str_(max_length=64), values=str_(max_length=64))
```

or already combined into `items` (which should yield key-value pairs):

```
dict_(items=two(str_(max_length=64)))
```

Those two styles are mutually exclusive - only one can be used at a time.

Parameters

- **keys** – Generator for dictionary keys
- **values** – Generator for dictionary values
- **items** – Generator for dictionary items (2-element tuples).
- **min_length** – A minimum number of items the resulting dictionary will contain
- **max_length** – A maximum number of items the resulting dictionary will contain

3.2.2 Combinators

If you want to have a generator that produces values of more than one type, use the simple *one_of()* function or the more sophisticated *frequency()* combinator.

For a simpler task of always choosing a value from a predefined set of objects, the *elements()* function will come handy.

pyqcy.arbitraries.combinators.**one_of**(**generators*)
Generator that yields values coming from given set of generators.

Generators can be passed either directly as arguments:

```
one_of(int, float)
```

or as a list:

```
one_of([int, float])
```

Every generator has equal probability of being chosen. If you need non-uniform probability distribution, use the *frequency()* function.

pyqcy.arbitraries.combinators.**frequency**(**distribution*)
Generator that yields coming from given set of generators, according to their probability distribution.

The distribution is just a set of tuples: `(gen, freq)` which can be passed either directly as arguments:

```
frequency((int, 1), (float, 2))
```

or a a list:

```
frequency([(int, 1), (float, 2)])
```

The second element of tuple (`freq`) is the relative frequency of values from particular generator, compared to those from other generators. In both examples above the resulting generator will yield `floats` twice as often as `ints`.

Typically, it's convenient to use floating-point frequencies that sum to `1.0` or integer frequencies that sum to `100`.

pyqcy.arbitraries.combinators.**elements**(**list*)
Generator that returns a random element from given set.

Elements can be passed either directly as arguments:

```
elements(1, 2, 3)
```

or as a list:

```
elements([1, 2, 3])
```

Every element has equal probability of being chosen.

Data structures

For testing higher level code, it is often required to prepare more complex input data and not just simple, uniform collections of elements. Even then, it can be possible to avoid writing a custom generator if we use the *data()* function.

pyqcy.arbitraries.combinators.**data**(*schema*)
Generator that outputs data structures conforming to given schema.

Parameters **schema** – A list or dictionary that contains either immediate values or other generators.

Note: `schema` can be recursive and combine lists with dictionaries into complex structures. You can have nested dictionaries, lists containing lists, dictionaries with lists as values, and so on.

A typical example of using *data()*:

```
import string

@qc
def creating_user_works(
    request=data({
        'login': str_(of=string.ascii_letters | string.digits,
                      min_length=3, max_length=32),
        'password': str_(min_length=8, max_length=128),
    })
):
    response = create_user(request['login'], request['password'])
    assert response['status'] == "OK"
```

Applying functions

Yet another way of combining generators is to use them as building blocks for whole object *pipelines*. This is possible thanks to *apply()* combinator.

pyqcy.arbitraries.combinators.**apply**(*func*, **args*, ***kwargs*)

Generator that applies a specific function to objects returned by given generator(s).

Any number of generators can be passed as arguments, and they can be both positional (`args`) or keyword arguments (`kwargs`). In either case, the same invocation style (i.e. positional or keyword) will be used when calling the `func` with actual values obtained from given generators.

As an example, the following call:

```
apply(json.dumps, dict_(items=two(str)))
```

will create a generator that yields results of `json.dumps(d)`, where `d` is an arbitrary dictionary that maps strings to strings.

Similarly, using *apply()* as shown below:

```
apply(itertools.product, list_(of=int), repeat=4)
```

gets us a generator that produces results of `itertools.product(l, repeat=4)`, where `l` is an arbitrary list of `int`s.

3.3 Running the tests

Once you have written some tests using *pyqcy*, you would most likely want run them.

If you already have a test suite of different kinds of tests for your projects (typically at least unit tests), you probably want to *integrate* *pyqcy* properties into that.

Alternatively, properties can be also verified using a built-in, standalone test runner.

3.3.1 Test runner

pyqcy includes a readily available test runner which can be used to run verification tests for all properties defined within given module. For it to work, you just need to include a traditional `if __name__ == '__main__':` boilerplate which calls `pyqcy.main()`:

```
from pyqcy import *

# ... define test properties here ...

if __name__ == '__main__':
    main()
```

This default test runner will go over all properties defined within this module, as well as all modules it imports, and execute tests for them. It is intentionally similar in usage to standard *unittest.main* and shares many parameters with the *unittest* runner (to the extent it makes sense for *pyqcy* tests, of course).

pyqcy.runner.**main**(*module='__main__'*, *exit=True*, *verbosity=2*, *failfast=False*)

Built-in test runner for properties.

When called, it will look for all properties (i.e. functions with `qc()` decorator) and run checks on them.

Arguments are intended to mimic those from `unittest.main()`. Return value is the total number of properties checked, provided `exit` is `False` and program doesn't terminate.

3.3.2 Integration with testing frameworks

If you are already using a unit testing framework, you can easily integrate *pyqcy* property tests into it.

For this, there is a `TestCase` class which is a descendant of the standard `unittest.TestCase`. Any test cases built upon it will be gathered and ran by pretty much any testing framework - be it `unittest` itself, *nose*, *py.test*, etc.

Therefore all we need to do is to put out properties inside a `TestCase` subclass:

```
from pyqcy import *

class Arithmetic(TestCase):
    @qc
    def addition_on_ints(x=int, y=int):
        assert isinstance(x + y, int)
    @qc
    def subtraction_on_ints(x=int, y=int):
        assert isinstance(x - y, int)
```

There is no need to rename the properties to start with `test_` but we should retain the `qc()` decorator. We also don't need to include any other methods that would explicitly run tests for our properties, as the base `TestCase` class will take care of it automatically.

class pyqcy.integration.**TestCase**(*methodName='runTest'*)

unittest test case for pyqcy properties.

Properties defined here within subclasses of `TestCase` will be verified automatically as a part of standard *unittest* run. To define them, use the typical syntax with `qc()` decorator:

```
    class Sorting(TestCase):
        '''Properties that must hold for a sorting.'''
        @qc
        def sort_preserves_length(
            l=list_(of=int, max_length=128)
        ):
            assert len(l) == len(list(sorted(l)))
        @qc
        def sort_finds_minimum(
            l=list_(of=int, min_length=1, max_length=128)
```

```
    ):
        assert min(l) == list(sorted(l))[0]
```

Since `TestCase` itself is a subclass of standard `unittest.TestCase`, it will be discovered by `unittest.main()`, *nosetests* or similar testing utilities.

3.4 Gathering statistics

As your tests are ran, you may want to gain some insight into what test cases are actually generated in order to verify your properties. Usually, however, there will be hundreds or thousands of them, so you certainly don't want to wade through them all.

To consolidate this data into more useful information, *pyqcy* provides you with statistical functions: *collect()* and *classify()*.

Warning: All statistical functions described below must be `yield` from within test properties to be recorded.

`pyqcy.statistics.`**`collect`**`(`*value*`)`

Collects test cases that share the same value (passed as argument) for statistical purposes.

Parameters **`value`** – Value to collect. This can be any hashable, i.e. a value that could be a set element or dictionary key.

Typical usage of *collect()* is as follows:

```
@qc
def sort_works(
    l=list_(int, min_length=1, max_length=100)
):
    yield collect(len(l))
    assert list(sorted(l))[0] == min(l)
```

Checking the above property will produce output similar to this:

```
sort_works: passed 100 tests.
1.00%: 1
1.00%: 2
...
1.00%: 100
```

`pyqcy.statistics.`**`classify`**`(`*condition*, *label*`)`

Classifies test cases depending on whether they satisfy given condition.

If a test case meets the condition, it will be "stamped" with given label that will subsequently appear in statistical report displayed after a property has been tested.

Parameters

- **`condition`** – Condition that the test data should satisfy in order for the test case to be stamped with `label`.
- **`label`** – A label to be associated with this test case if `condition` turns out to be true

Typical usage is as follows:

```
@qc
def sort_preserves_length(
    l=list_(int, min_length=1, max_length=100)
```

```
    ):
        yield classify(len(l) == 0, "empty list")
        yield classify(len(l) < 10, "short list")
        assert len(list(sorted(l))) == len(l)
```

Checking the above property will produce something like the following output:

```
sort_preserves_length: passed 100 tests.
1.00%: empty list, short list
9.00%: short list
90.00%: <rest>
```