An Introduction to PyBT

Vivien Rindisbacher

April 2023

1 Links

- PyPI
- Github

2 Introduction

Unit Testing, Test Driven Development, Acceptance Testing, End to End Testing, Integration Testing...

These days, there are so many methods proposed for testing code it is surprising that writing correct code remains so difficult. In this introduction to PyBT, a library for property-based testing in Python, I introduce the common flaw of many testing methodologies and show how property-based testing circumvents the issue.

3 How We Write Code

Programming is hard. This property stems from a simple phenomenon - programming is abstract. Here is a simple example that illustrates the problem.

Write me a program that sorts a list of integers, floats, and booleans.

Of course, it is not difficult to imagine *what it means* to sort a list. However, when writing such a program, one will never deal with this problem by thinking of some abstract list. Instead, one will think of a series of examples helpful in creating such a program. For example...

```
[1, 2, 5, 4] -> [1, 2, 4, 5]
[1.5, 3.2, 2.3, 6.5, 5.3] -> [1.5, 2.3, 3.2, 5.3, 6.5]
[True, False, True, True] -> [False, True, True, True]
...
```

There is nothing inherently wrong with programming this way. In fact, it would be very difficult to reason about any program without examples. However, our reasoning creates a pseudo-tautology when it comes to testing.

4 Why Does Abstraction Matter

The programmer wrote code by reasoning over some examples. The common approach to testing would be the same! Encode some possibilities for l and *sort_list(l)*, and assert that they match.

Clearly, our test cases will be similar to the cases used in writing the function in the first place. This is bad.

One might think that programmers need only challenge themselves in writing tests, by thinking of edge cases or complex examples that make their code fail. There are two unfortunate issues with this idea.

First, the universe of possible inputs is infinite, and therefore there is no way one would find all the examples that cause their code to fail. Second, thinking of these examples, reasoning about the correct output, and encoding them is immensely time-consuming.

5 Testing Abstractly

We have established that programs are abstract. Therefore, a natural next step is to test code abstractly. Enter property-based testing.

A good question to ask at this point is, *If I had to prove that sort_list is correct, what would I actually be proving?*. Theoretically, we need a universally quantified proposition, which is a statement of correctness for *sort_list*. In layman's terms, we need to find a property that describes what it means for *sort_list* to be correct. Here is a proposal...

Theorem sort_correct: $\forall\, l\ i\ j, sorted = sort\_list(l) \rightarrow i < size(l) \rightarrow j < i \rightarrow sorted[j] \leq sorted[i]$

This reads: *for all l, i, and j, if sorted is the result of calling sort_list on l, i is less than the size of l, and j is less than i, then the element at index j of sorted will be less than or equal to the element at index i of sorted.*

This is not too bad! Of course, if every element to the left of some element a is less than a, then the list is sorted. If we could test this property, over many l, we would be in a much better place.

One challenge is imagining how this proposition could be encoded in a programming language like Python. However, if we could do this, then we would be able to generate as many test cases (short of infinite) as we want and see if the proposition or property holds. Here is a proposed way of encoding this.

```
def sort_list_test(l : list[int] | list[float] | list[bool]):
    sorted = sort_list(l)
    for i in range(len(sorted)):
        for j in range(i):
            assert (sorted[j] <= sorted[i])
```

This function matches our proposition exactly. Specifically, our parameter is our universally quantified variable list l. We then encode that all indexes i and j are such that $j < i$ and $i < size(l)$. Finally, we assert that $sorted[j] \leq sorted[i]$.

Now, if we generated 10000 random test cases for *sort_list_test*, we would have high assurance that our function is correct. This is exactly what property-based testing does.

6 PyBT

It is not too difficult to encode our properties as functions. However, we still need to generate random inputs based on the types of our function arguments. This is what PyBT does.

6.1 Main Functionality

The library provides a decorator named *pybt*. The arguments to this decorator are...

- n: the number of tests to run (defaults to 1000).
- *generators*: user-provided generators to be used to generate function arguments (defaults to None).
- *hypotheses*: hypotheses that generated arguments are constrained by.
- *max_basic_arg_size*: maximum size of string, ints, floats, etc.
- *max_complex_arg_size*: maximum size to use for complex structures like list an dict. Also applies to the depth of random types generated by any.

6.2 Types Supported

PyBT supports basic types, union types, and nested types. It supports *list*, *dict*, *int*, *str*, *float*, *bool*. It also supports *any*, which will generate a random type, and, subsequently, arguments of that type. The types generated by any are constrained to *list*, *dict*, *int*, *str*, *float*, and *bool*.

There are plans to add random generation of classes and callables in the future. It is always possible to test functions with types not supported by PyBT. To do this, one needs only use custom generators. You can see an example below, in the section Generators.

6.3 Usage

Let's use our *sort_list* example from above.

```
from pybt.core.core import pybt
from unittest import TestCase

class TestSorted(TestCase):
    pybt_small = pybt(max_complex_arg_size=5, max_basic_arg_size=10000)

    @pybt_small
    def sort_list_test(self, l : list[int] | list[float] | list[bool]):
        sorted = sort_list(l)
        for i in range(len(sorted)):
            for j in range(i):
                assert (sorted[j] <= sorted[i])
```

First, we import the *pybt* decorator from PyBT and the *TestCase* class from the unittest module. Then, we declare our tests in a way that is standard in Python (using unittest). The caveat here is that our test is a property and that we have added the decorator *pybt_small*. The unittest framework will pick up *sort_list_test*, and we will be able to run it from our IDE. When run, *pybt_small* will generate 1000 random inputs to test this property.

6.4 Hypotheses

If you need to constrain the arguments that are passed to your function, you can constrain them using hypotheses. For example, if we wanted every single integer that appeared in l to be less than 2 for *sort_list*, we would specify that in the following way:

```
def constrain_l(l : list):
    for el in l:
        if type(el) == int:
            if el > 2:
                return False
    return True


hypotheses = {
    "l" : lambda l : constrain_l(l)
```

```
}


pybt_small = pybt(max_complex_arg_size=5, max_basic_arg_size=10000, hypotheses=hypotheses)
```

6.5 Generators

If you have arguments that should be generated in a very specific way, then you can provide your own generators, through the *generators* argument. For example, let's say we only wanted our *sort_list* function to run on lists of ints that are coerced to strings. We could do so in the following way.

```
import random


def gen_l():
    l = []
    for _ in range(5):
        el = random.randint(1000)
        l.append(str(el))
    return l


generators = {
    "l" : gen_l
}


pybt_small = pybt(max_complex_arg_size=5, max_basic_arg_size=1000, generators=generators)
```

7 Beware

It is very important that the property used matches the code it is intended for. To clarify what is meant by this, consider this example of *sort_list*.

```
def sort_list(l):
    return []
```

Here, the property we used before will obviously hold true, but this is not sorting a list. There are all sorts of gotcha's that can happen when coming up with a specification. Thus, it is important that you think very carefully about the property you use.

8 Conclusion

Property-based testing presents a shift from writing to test cases, to reasoning (abstractly) about what correctness means. Property-based testing is not a replacement for any testing methodology, but it does work in conjunction with most. For example, test-driven development using property-based testing would be an extremely powerful tool.