Quick Start to Hypothesis

Hypothesis is a Python testing library which we'll use occasionally in this course for exercises and assignments. If you haven't installed Hypothesis already, please make sure to do so by following Part 4 of the Software Guide.

When writing tests, we often try to identify key properties on the inputs to the function being tested, and then pick representative inputs that meet these properties, and use these inputs to write tests. We can extend this idea to trying to identify key properties of the function itself: central relationships between their inputs and outputs that must hold for all possible inputs. This type of testing is called property-based testing, and the most famous implementation of this type of testing in Python is the hypothesis library.

Let's see a concrete example of what these "properties" might be.

Exercise 1 example

Consider the task assigned for Exercise 1, which has a class with the following two methods:

```
class SuperDuperManager:
    def add_car(self, id_, fuel):
        """Add a new car to the system.

        The new car is identified by the string <id_>, and has initial amount
        of fuel <fuel>.

        Do nothing if there is already a car with the given id.

        @type self: SuperDuperManager
        @type id_: str
        @type fuel: int
        @rtype: None
        """
```

One property of `add_car` that we might assert is that it always return `None`, which should be true regardless of its input.

We could pick some random values to call `add_car` on manually, in a standard unit test:

```
import unittest
from ex1 import SuperDuperManager


class TestCar(unittest.TestCase):
    def test_add_car_returns_None(self):
        """Check that add_car always returns None."""
        manager = SuperDuperManager()
        return_value = manager.add_car('my cool car', 100)
        self.assertIsNone(return_value)
```

But there is nothing "special" about the car id and fuel amount we chose, and ideally we would test this property of `add_car` for multiple values. Rather than writing out all these tests by hand, we make use of Hypothesis' ability to generate random values for us, and repeat a single test multiple times on all of these random values.

Here's how this would look:

```
import unittest
from hypothesis import given
from hypothesis.strategies import integers, text
from ex1 import SuperDuperManager


class TestCarWithHypothesis(unittest.TestCase):
    @given(text(), integers(min_value=0))
    def test_add_car_returns_None(self, id_, fuel):
        """Check that add_car always returns None."""
        manager = SuperDuperManager()
        return_value = manager.add_car(id_, fuel)
        self.assertIsNone(return_value)
```

The test actually is structured very similarly, with the exception of the line

```
    @given(text(), integers(min_value=0))
```

What this does is say that the following test will take two additional arguments, a random string and a random integer greater than or equal to 0. We use the Hypothesis function `text()` to generate a random string, and `integers()` to generate a random integer.

When you run this test, you might notice it takes a bit longer to run: that's because Hypothesis actually repeats this test several times, choosing new random inputs each time! By leveraging the power of this random input generation, we can write short tests that check for very general properties of our function behaviour.

One more example When we start writing more complex code, we start wanting to think not just about testing functions in isolation, but their interactions with each other. At this point, it is extremely helpful to try thinking about the properties that govern this interaction.

For example, our SuperDuperManager class has the following method:

```
    def get_car_fuel(self, id_):
        """Return the amount of fuel of the car with the given id.

        Return None if there is no car with the given id.

        @type self: SuperDuperManager
        @type id_: str
        @rtype: int | None
       """
```

One interaction between this method and the `add_car` method from above is that when we add a new car with the specified amount of fuel, we should see that same fuel returned when we call `get_car_fuel`. We can capture this property very nicely with the following hypothesis test:

```
    @given(text(), integers(min_value=0))
    def test_car_has_initial_fuel_set(self, id_, fuel):
        """Check that a car's initial fuel is set properly.
        """
        manager = SuperDuperManager()
        manager.add_car(id_, fuel)                          # Create new car
        self.assertEqual(fuel, manager.get_car_fuel(id_))  # Check new car fuel
```

The setup for the test is the same as before: we have a random id and random amount of fuel given, and we call `add_car` and then `get_car_fuel`, expecting the amount of fuel returned to be the same as the initial amount.

Further reading Hypothesis is a power property-based testing library, and we're only scratching the surface of it here. If you'd like more information, please consult the official Hypothesis documentation.