Caerbannog Documentation

Release 0.1.1

AlexV

Apr 03, 2019

Contents:

Follow the white rabbit... at your own peril.

CHAPTER 1

Object Oriented Turtle

Data and behavior are combined into one object. Client calls a turtle class instance. Turtle class needs to keep track of the Turtle 'State'.

The state is defined here for simplicity as (position, angle, pen's state - up or down).

In this design, the turtle class holds the turtle state and mutates it. This makes it quite simple to wrap the python turtle API and even "extend" it.

Our tootle class has a few methods : - move(distance) - left(angle) - right(angle) - penup() - pendown()

Side note : we use pint for unit of measure (radians / degrees)

1.1 Client code

The client code is quite simple. We create a Tootle instance named 'tt'

Listing 1: ../01/client.py

```
def triangle():
    tt = tootle.Tootle()
    dist = 200
```

And we call Tootle's methods to draw a triangle

Listing 2: ../01/client.py

```
def triangle():
    tt = tootle.Tootle()
    dist = 200

    tt.move(dist)
    tt.left(120)
    tt.move(dist)
```

(continues on next page)

(continued from previous page)

```
    tt.left(120)
    tt.move(dist)
```

1.2 Tootle code

The tootle code is usual object oriented code

We use an enum for the two possible penstate values, as well as having penstate as a type.

Listing 3: ../01/tootle.py

```
# We use enum for simple values as type
class PenState(enum.Enum):
    UP = -1
    DOWN = 1
```

We also have a Tootle class, wrapping the existing python Turtle class. It exposes the state as members of this class.

Listing 4: ../01/tootle.py

```
# The usual OO inheritance interface
# taking turtle.Turtle as an unknown black box.
# We provide shortcuts to some of its methods and attributes here for interfacing with turtle,
# while trying to keep the turtle API small.
class Tootle(turtle.Turtle):
    """
    Inheriting from provided Turtle class, OO style.
    >>> t = Tootle()


    """
    @property
    def position(self):
        return super().position() * ureg.pixel

    @property
    def angle(self):
        return super().heading() * ureg.degrees

    @property
    def penState(self):
        return super().pen().get('pendown')

    def move(self, distance: int):
        super().forward(distance=distance)

    def right(self, angle: int):
        super().right(angle * ureg.degrees)

    def left(self, angle: int):
        super().left(angle * ureg.degrees)

    def penup(self):
```

(continues on next page)

(continued from previous page)

```
        super().penup()

    def pendown(self):
        super().pendown()
```

It also provides methods matching our API, that are used to mutate the state (via the python Turtle instance).

Listing 5: ../01/tootle.py

```
# The usual OO inheritance interface
# taking turtle.Turtle as an unknown black box.
# We provide shortcuts to some of its methods and attributes here for interfacing with turtle,
# while trying to keep the turtle API small.
class Tootle(turtle.Turtle):
    """
    Inheriting from provided Turtle class, OO style.
    >>> t = Tootle()


    """
    @property
    def position(self):
        return super().position() * ureg.pixel

    @property
    def angle(self):
        return super().heading() * ureg.degrees

    @property
    def penState(self):
        return super().pen().get('pendown')

    def move(self, distance: int):
        super().forward(distance=distance)

    def right(self, angle: int):
        super().right(angle * ureg.degrees)

    def left(self, angle: int):
        super().left(angle * ureg.degrees)

    def penup(self):
        super().penup()

    def pendown(self):
        super().pendown()
```

Notice also how the code uses docstrings and doctests to certify the documentation stays uptodate.

If you know python, this code should have no mystery for you.

1.3 Tootle test

Since we want our code to work, we need to test it.

First we do not want to display a graphical "TurtleScreen" everytime we run a test. This is quite tricky to do, and for now we will rely on a monkey patch. TODO

Then we define our test class, using the unittest core library

Listing 6: ../01/test_tootle.py

```
    since the inheritance mechanism uses the logic of the super class and related class hierarchies.
    Also any action with side effect will modify the environment, which we don't control by definition,
    by changing the start position of the turtle and will affect the following test, for example.

    What we can do however, is that our own code is using the rest of the code as expected, using a mock.
    The mock will prevent any side effect by intercepting any procedure call, and register counters,
    before it interferes with the outside world. but we need to investigate the superclass hierarchy to understand
    its behavior and relationship.This breaks the original encapsulation intent...

    Anyway, it is better than not testing, but this tests almost nothing.
    It will not test the whole behavior, and it doesn't scale to test integration of multiple components.
    It is also quite fragile with inheritance, as we have to plug our mock in the exact place, which can be quite hacky
    """

    @mock.patch('turtle.TurtleScreen', autospec=True)
    def setUp(self, mock_tscreen):
        self.t = tootle.Tootle()
        # TODO ... currently mock breaks the turtle...
        assert mock_tscreen.called_with()

    def check_move(self, dist: int):
        """
        Implements one check of move
        :param dist: the distance to move
        :return:
        """
        # get position before
        p0 = self.t.position

        # get position after
        p1 = self.t.position

        assert p1 - p0 == dist

    @mock.patch("turtle.Turtle.forward")
    def test_move(self, mockturtle):
        """ Testing multiple values of distance"""
        for d in [random.randint(0, self.max_test_dist) for _ in range(20)]:
```

You can notice how we have a few '**check_**' functions that verify the behavior of a Turtle method for a specific parameter.

Listing 7: ../01/test_tootle.py

```
    since the inheritance mechanism uses the logic of the super class and related class hierarchies.
    Also any action with side effect will modify the environment, which we don't control by definition,
    by changing the start position of the turtle and will affect the following test, for example.

    What we can do however, is that our own code is using the rest of the code as expected, using a mock.
    The mock will prevent any side effect by intercepting any procedure call, and register counters,
    before it interferes with the outside world. but we need to investigate the superclass hierarchy to understand
    its behavior and relationship.This breaks the original encapsulation intent...

    Anyway, it is better than not testing, but this tests almost nothing.
    It will not test the whole behavior, and it doesn't scale to test integration of multiple components.
    It is also quite fragile with inheritance, as we have to plug our mock in the exact place, which can be quite hacky
    """

    @mock.patch('turtle.TurtleScreen', autospec=True)
    def setUp(self, mock_tscreen):
        self.t = tootle.Tootle()
        # TODO ... currently mock breaks the turtle...
        assert mock_tscreen.called_with()

    def check_move(self, dist: int):
        """
        Implements one check of move
        :param dist: the distance to move
        :return:
        """
        # get position before
        p0 = self.t.position

        # get position after
        p1 = self.t.position

        assert p1 - p0 == dist

    @mock.patch("turtle.Turtle.forward")
    def test_move(self, mockturtle):
        """ Testing multiple values of distance"""
        for d in [random.randint(0, self.max_test_dist) for _ in range(20)]:
```

Then we have a few 'test_method' that actually run these checks for a larger number of parameters. It helps us have a better coverage of the data space.

Listing 8: ../01/test_tootle.py

```
    since the inheritance mechanism uses the logic of the super class and related class hierarchies.
    Also any action with side effect will modify the environment, which we don't control by definition,
```

(continues on next page)

(continued from previous page)

```
    by changing the start position of the turtle and will affect the following test, for example.

    What we can do however, is that our own code is using the rest of the code as expected, using a mock.
    The mock will prevent any side effect by intercepting any procedure call, and register counters,
    before it interferes with the outside world. but we need to investigate the superclass hierarchy to understand
    its behavior and relationship.This breaks the original encapsulation intent...

    Anyway, it is better than not testing, but this tests almost nothing.
    It will not test the whole behavior, and it doesn't scale to test integration of multiple components.
    It is also quite fragile with inheritance, as we have to plug our mock in the exact place, which can be quite hacky
    """

    @mock.patch('turtle.TurtleScreen', autospec=True)
    def setUp(self, mock_tscreen):
        self.t = tootle.Tootle()
        # TODO ... currently mock breaks the turtle...
        assert mock_tscreen.called_with()

    def check_move(self, dist: int):
        """
        Implements one check of move
        :param dist: the distance to move
        :return:
        """
        # get position before
        p0 = self.t.position

        # get position after
        p1 = self.t.position

        assert p1 - p0 == dist

    @mock.patch("turtle.Turtle.forward")
    def test_move(self, mockturtle):
        """ Testing multiple values of distance"""
        for d in [random.randint(0, self.max_test_dist) for _ in range(20)]:
```

Despite these efforts to test our turtle, there are many cases that we did not consider. For example : - We test only from the initial turtle state, but what if the method behavior depends on the turtle state ? - We test only a very small set of possible values, what is the method behavior is different for some untested value ? - How will be the behavior for “unexpected” inputs ? Will it crash ? throw and exception ? which one ? We do not test the unexpected (obviously)

These are the current limitation of basic python tests. We will attempt to improve our test coverage in the following chapters.

1.4 Conclusion

Pros:

- Familiar for most python programmers.

Cons :

- Stateful. Hard to test methods independently from the state.
- Inheritance effectively provides a blackbox, that can be hard to setup for tests.
- Cannot compose between method calls, we have to actually do one behavior at a time.
- Hard coded dependencies in module, the client needs to bring in everything.

1.5 API

Tootle API

CHAPTER 2

Abstract Data Turtle

Data is separate from behaviour.

Data structure is mutable, private, and only the turtle functions can change the data From hte caller point of view it is an opaque data structure

Each function needs the state to be passed in.

```
import turtle
import enum
import pint

ureg = pint.UnitRegistry()


# Side Note: Yes, Python has enums !
class PenState(enum.Enum):
    UP = -1
    DOWN = 1


# Simplest interface for clarity, but can only support one turtle.
# We use this global as a way to simplify functions interface to keep clarity.
_abstle = None


# Using class as data structure, for typing, and '_' as the usual python convention␣
↪for "private".
# Remember, Types were used before Object Oriented programming...
# The usual OO delegation interface
# taking turtle.Turtle as an unknown black box.
# Composition, not inheritance.
class _TurtleState:
    """
    TurtleState is mutable and therefore private.
    """
```

(continues on next page)

(continued from previous page)

```
    def __init__(self):
        self.update()

    def update(self):
        global _abstle

        """Update from the global abstle"""
        self.position = _abstle.position()
        self.angle = _abstle.heading()
        self.penState = _abstle.pen().get("pendown")


def create(supermodel=None):
    global _abstle
    _abstle = supermodel if supermodel is not None else turtle.Turtle()
    return _TurtleState()


def move(distance: int, state: _TurtleState):
    global _abstle

    # TMP HACK
    distance = int(distance)

    _abstle.forward(distance=distance)

    # This will mutate the state
    state.update()


def right(angle: int, state: _TurtleState):
    global _abstle

    # TMP HACK
    angle = int(angle * ureg.degrees)

    _abstle.right(angle)

    # This will mutate the state
    state.update()


def left(angle: int, state: _TurtleState):
    global _abstle

    # TMP HACK
    angle = int(angle)

    _abstle.left(angle)

    # This will mutate the state
    state.update()


def penup(state: _TurtleState):
    global _abstle
```

(continues on next page)

(continued from previous page)

```
    _abstle.penup()
    # This will mutate the state
    state.update()


def pendown(state: _TurtleState):
    global _abstle
    _abstle.pendown()
    # This will mutate the state
    state.update()
```

Pros:

- simple to implement
- cant do inheritance -> forces composition

Cons:

- Stateful/Blackbox/hard to test

2.1 Pythonic

2.2 API

Abstle API

CHAPTER 3

Functional Turtle

Data is immutable

Client has to create the state and pass it into a functio and get the new state back -> The client is involved in managing the state

Functions now return the state, and can be composed !

3.1 Pythonic

3.2 Pure

3.3 API

Functle API

3.4 Overall

Pros: - Immutability : easier to reason about - Stateless: Easier to test - Fcuntions are composable

Cons: - client has to keep track fo the state - Hardc oded dependencies

CHAPTER 4

State Monad

State threading behind the scene.

4.1 Problem upgrade

Lets say the turtle cant go after a certain distance. We need to return the distance actually moved. -> Passing the state around is complex, and we can't compose easily as before.

4.2 Currying

We transform the functions. Instead of having one two params function, we get two, one-param, function. We now say that the state is a wrapper around the function returned from a function.

4.3 Pythonic

4.4 Pure

4.5 Monadic

4.6 API

Montle API

Pros: - looks imperatve but preserve immutability - functions are still composable

Cons: - harder to implement and use

CHAPTER 5

Various API documentation

5.1 Tootle API

class tootle.**PenState**
An enumeration.

5.2 Abstle API

class abstle.**PenState**
An enumeration.

5.3 Functle API

class functle.**PenState**
An enumeration.

class functle.**TurtleState**
Immutable public State.

angle
Alias for field number 1

pen
Alias for field number 2

position
Alias for field number 0

functle.**curry**(*fun*)
Curry the function : changes some parameters (the ones not passed at first) into a later function call. Effectively splits one call into two. >>> def add(a,b): ... return a + b >>> addfirst = curry(add) >>> andthen = addfirst(2) >>> addthen(3) 5

functle.**functle**()
A context in which a turtle is available, along with its state, and a functional application accumulator

functle.**schoenfinkel**(*fun*)
Curry the function : changes some parameters (the ones not passed at first) into a later function call. Effectively splits one call into two. >>> def add(a,b): ... return a + b >>> addfirst = curry(add) >>> andthen = addfirst(2) >>> addthen(3) 5

5.4 Montle API

class montle.**PenState**
An enumeration.

class montle.**TurtleImpl**
Basic class to hold implementation and state together For simplicity sake

impl
Alias for field number 1

state
Alias for field number 0

class montle.**TurtleState**
Immutable public State.

angle
Alias for field number 1

pen
Alias for field number 2

position
Alias for field number 0

montle.**curry**(*fun*)
Curry the function : changes some parameters (the ones not passed at first) into a later function call. Effectively splits one call into two. >>> def add(a,b): ... return a + b >>> addfirst = curry(add) >>> andthen = addfirst(2) >>> addthen(3) 5

montle.**montle**() → TurtleImpl
A context in which a turtle is available, as well as its state

montle.**schoenfinkel**(*fun*)
Curry the function : changes some parameters (the ones not passed at first) into a later function call. Effectively splits one call into two. >>> def add(a,b): ... return a + b >>> addfirst = curry(add) >>> andthen = addfirst(2) >>> addthen(3) 5

montle.**uncurry**(*fun*)
Uncurry the function : changes some :param fun: :return:

CHAPTER 6

Indices and tables

- genindex
- modindex
- search

Python Module Index

Index