Table of Contents

Test Package Documentation

This package provides testing utilities and assertion functions for MoonBit prog
rams. It includes functions for comparing values, checking object identity, and
creating structured test outputs with snapshot testing capabilities.

Basic Test Structure

MoonBit tests are written using the test keyword:

```

  test "basic test example" {
    let result = 2 + 2
    inspect(result, content="4")


  }
```

Assertion Functions

Object Identity Testing

Test whether two values refer to the same object in memory:

```

  test "object identity" {
    let str1 = "hello"
    let _str2 = "hello"
    let str3 = str1


    @test same_object(str1, str3)









    let arr1 = [1, 2, 3]
    let _arr2 = [1, 2, 3]
    let arr3 = arr1
    @test same_object(arr1, arr3)
  }
```

Failure Testing

Explicitly fail tests with custom messages:

```
1
2   test "conditional failure" {
3     let value = 10
4     if value < 0 {
5       @test fail("Value should not be negative: \{value}")
6     }
7
8
9     inspect(value, content="10")
10  }
```

Test Output and Logging

Create structured test outputs using the Test type:

```
1
2   test "test output" {
3     let t = @test new("Example Test")
4
5
6     t write("Testing basic functionality: ")
7     t writeln("PASS")
8
9
10    t writeln("Step 1: Initialize data")
11    t writeln("Step 2: Process data")
12    t writeln("Step 3: Verify results")
13
14
15  }
```

Snapshot Testing

Compare test outputs against saved snapshots:

```
1
2   test "snapshot testing" {
3     let t = @test new("Snapshot Test")
4
5
6     t writeln("Current timestamp: 2024-01-01")
7     t writeln("Processing items: [1, 2, 3, 4, 5]")
8     t writeln("Result: SUCCESS")
9
10
11
12    t snapshot(filename="test_output")
13  }
```

Advanced Testing Patterns

Testing with Complex Data

Test functions that work with complex data structures:

```

  test "complex data testing" {

    let numbers = [1, 2, 3, 4, 5]
    let doubled = numbers map(fn(x) { x * 2 })
    inspect(doubled, content="[2, 4, 6, 8, 10]")


    let person_data = ("Alice", 30)
    inspect(person_data 0, content="Alice")
    inspect(person_data 1, content="30")
  }
```

Error Condition Testing

Test that functions properly handle error conditions:

```

  test "error handling" {
    fn safe_divide(a : Int, b : Int) -> Int? {
      if b == 0 {
        None
      } else {
        Some(a / b)
      }
    }


    let result = safe_divide(10, 2)
    inspect(result, content="Some(5)")


    let error_result = safe_divide(10, 0)
    inspect(error_result, content="None")
  }
```

Property-Based Testing

Test properties that should hold for various inputs:

```
1
2   test "property testing" {
3     fn is_even(n : Int) -> Bool {
4       n % 2 == 0
5     }
6
7
8     let test_values = [0, 2, 4, 6, 8, 10]
9     for value in test_values {
10      if not(is_even(value)) {
11        @test fail("Expected \{value} to be even")
12      }
13    }
14
15
16    let odd_values = [1, 3, 5, 7, 9]
17    for value in odd_values {
18      if is_even(value) {
19        @test fail("Expected \{value} to be odd")
20      }
21    }
22  }
```

Test Organization

Grouping Related Tests

Use descriptive test names to group related functionality:

```
1
2   test "string operations - concatenation" {
3     let result = "hello" + " " + "world"
4     inspect(result, content="hello world")
5   }
6
7
8   test "string operations - length" {
9     let text = "MoonBit"
10    inspect(text length(), content="7")
11  }
12
13
14  test "string operations - substring" {
15    let text = "Hello, World!"
16    let sub = text length()
17    inspect(sub, content="13")
18  }
```

Setup and Teardown Patterns

Create helper functions for common test setup:

```

  test "with setup helper" {
    fn setup_test_data() -> Array[Int] {
      [10, 20, 30, 40, 50]
    }

    fn cleanup_test_data(_data : Array[Int]) -> Unit {

    }

    let data = setup_test_data()


    inspect(data.length(), content="5")
    inspect(data[0], content="10")
    inspect(data[4], content="50")
    cleanup_test_data(data)
  }
```

Testing Best Practices

Clear Test Names

Use descriptive names that explain what is being tested:

```

  test "user_can_login_with_valid_credentials" {

  }


  test "login_fails_with_invalid_password" {

  }


  test "shopping_cart_calculates_total_correctly" {

  }
```

One Concept Per Test

Keep tests focused on a single concept:

```
1
2
3   test "array_push_increases_length" {
4     let arr = Array::new()
5     let initial_length = arr length()
6     arr push(42)
7     let new_length = arr length()
8     inspect(new_length, content="\{initial_length + 1}")
9   }
10
11
12
13  test "array_push_adds_element_at_end" {
14    let arr = Array::new()
15    arr push(10)
16    arr push(20)
17    inspect(arr[arr length() - 1], content="20")
18  }
```

Use Meaningful Test Data

Choose test data that makes the test's intent clear:

```
1
2   test "tax_calculation_for_standard_rate" {
3     let price = 100
4     let tax_rate = 8
5     let calculated_tax = price * tax_rate / 100
6     inspect(calculated_tax, content="8")
7   }
```

Integration with MoonBit Build System

Tests are automatically discovered and run by the MoonBit build system:

- Use moon test to run all tests
- Use moon test --update to update snapshots
- Tests in *_test.mbt files are blackbox tests
- Tests in regular .mbt files are whitebox tests

Common Testing Patterns

- **Arrange-Act-Assert**: Set up data, perform operation, verify result
- **Given-When-Then**: Given some context, when an action occurs, then verify outcome
- **Red-Green-Refactor**: Write failing test, make it pass, improve code
- **Test-Driven Development**: Write tests before implementation

Performance Considerations

- Keep tests fast by avoiding expensive operations when possible
- Use setup/teardown functions to share expensive initialization
- Consider using smaller datasets for unit tests
- Save integration tests with large datasets for separate test suites

The test package provides essential tools for ensuring code quality and correctn
ess in MoonBit applications through comprehensive testing capabilities.