Table of Contents

Builtin Package Documentation

This package provides the core built-in types, functions, and utilities that are fundamental to MoonBit programming. It includes basic data structures, iterators, assertions, and core language features.

Core Types and Functions

Assertions and Testing

MoonBit provides built-in assertion functions for testing:

```
1
2   test "assertions" {
3
4     assert_eq(1 + 1, 2)
5     assert_eq("hello", "hello")
6
7
8     assert_true(5 > 3)
9     assert_false(2 > 5)
10
11
12    assert_not_eq(1, 2)
13    assert_not_eq("foo", "bar")
14  }
```

Inspect Function

The inspect function is used for testing and debugging:

```
1
2   test "inspect usage" {
3     let value = 42
4     inspect(value, content="42")
5     let list = [1, 2, 3]
6     inspect(list, content="[1, 2, 3]")
7     let result : Result[Int, String] = Ok(100)
8     inspect(result, content="Ok(100)")
9   }
```

Result Type

The Result[T, E] type represents operations that can succeed or fail:

```
1
2   test "result type" {
3     fn divide(a : Int, b : Int) -> Result[Int, String] {
4       if b == 0 {
5         Err("Division by zero")
6       } else {
7         Ok(a / b)
8       }
9     }
10
11
12    let result1 = divide(10, 2)
13    inspect(result1, content="Ok(5)")
14
15
16    let result2 = divide(10, 0)
17    inspect(result2, content="Err(\"Division by zero\")")
18
19
20    match result1 {
21      Ok(value) => inspect(value, content="5")
22      Err(_) => inspect(false, content="true")
23    }
24  }
```

Option Type

The Option[T] type represents values that may or may not exist:

```
1
2   test "option type" {
3     fn find_first_even(numbers : Array[Int]) -> Int? {
4       for num in numbers {
5         if num % 2 == 0 {
6           return Some(num)
7         }
8       }
9       None
10    }
11
12
13    let result1 = find_first_even([1, 3, 4, 5])
14    inspect(result1, content="Some(4)")
15
16
17    let result2 = find_first_even([1, 3, 5])
18    inspect(result2, content="None")
19
20
21    match result1 {
22      Some(value) => inspect(value, content="4")
23      None => inspect(false, content="true")
24    }
25  }
```

Iterator Type

The Iter[T] type provides lazy iteration over sequences:

```
1
2   test "iterators" {
3
4     let numbers = [1, 2, 3, 4, 5]
5     let iter = numbers iter()
6
7
8     let collected = iter collect()
9     inspect(collected, content="[1, 2, 3, 4, 5]")
10
11
12    let doubled = numbers iter() map(fn(x) { x * 2 }) collect()
13    inspect(doubled, content="[2, 4, 6, 8, 10]")
14
15
16    let evens = numbers iter() filter(fn(x) { x % 2 == 0 }) collect()
17    inspect(evens, content="[2, 4]")
18
19
20    let sum = numbers iter() fold(init=0, fn(acc, x) { acc + x })
21    inspect(sum, content="15")
22  }
```

Array and FixedArray

Built-in array types for storing collections:

```
1
2   test "arrays" {
3
4     let arr = Array::new()
5     arr push(1)
6     arr push(2)
7     arr push(3)
8     inspect(arr, content="[1, 2, 3]")
9
10
11    let fixed_arr = [10, 20, 30]
12    inspect(fixed_arr, content="[10, 20, 30]")
13
14
15    let length = fixed_arr length()
16    inspect(length, content="3")
17    let first = fixed_arr[0]
18    inspect(first, content="10")
19  }
```

String Operations

Basic string functionality:

```

  test "strings" {
    let text = "Hello, World!"


    let len = text length()
    inspect(len, content="13")


    let greeting = "Hello" + ", " + "World!"
    inspect(greeting, content="Hello, World!")


    let equal = "test" == "test"
    inspect(equal, content="true")
  }
```

StringBuilder

Efficient string building:

```

  test "string builder" {
    let builder = StringBuilder::new()
    builder write_string("Hello")
    builder write_string(", ")
    builder write_string("World!")
    let result = builder to_string()
    inspect(result, content="Hello, World!")
  }
```

JSON Support

Basic JSON operations:

```

  test "json" {

    let json_null = null
    inspect(json_null, content="Null")
    let json_bool = true to_json()
    inspect(json_bool, content="True")
    let json_number = (42 : Int) to_json()
    inspect(json_number, content="Number(42)")
    let json_string = "hello" to_json()
    inspect(
      json_string,
      content=(
        #|String("hello")
      ),
    )
  }
```

Comparison Operations

Built-in comparison operators:

```

  test "comparisons" {

    inspect(5 == 5, content="true")
    inspect(5 != 3, content="true")


    inspect(3 < 5, content="true")
    inspect(5 > 3, content="true")
    inspect(5 >= 5, content="true")
    inspect(3 <= 5, content="true")


    inspect("apple" < "banana", content="true")
    inspect("hello" == "hello", content="true")
  }
```

Utility Functions

Helpful utility functions:

```

  test "utilities" {

    let value = 42
    ignore(value)


    let result = not(false)
    inspect(result, content="true")


    let arr1 = [1, 2, 3]
    let arr2 = [1, 2, 3]
    let same_ref = arr1
    inspect(physical_equal(arr1, arr2), content="false")
    inspect(physical_equal(arr1, same_ref), content="true")
  }
```

Error Handling

Basic error handling with panic and abort:

```

  test "error handling" {

    fn safe_divide(a : Int, b : Int) -> Int {
      if b == 0 {


        0
      } else {
        a / b
      }
    }

    let result = safe_divide(10, 2)
    inspect(result, content="5")
    let safe_result = safe_divide(10, 0)
    inspect(safe_result, content="0")
  }
```

Best Practices

- **Use assertions liberally in tests**: They help catch bugs early and document expec
ted behavior
- **Prefer** Result over exceptions: For recoverable errors, use Result[T, E] instead
of panicking
- **Use** Option for nullable values: Instead of null pointers, use Option[T]
- **Leverage iterators for data processing**: They provide composable and efficient da
ta transformations
- **Use** StringBuilder for string concatenation: More efficient than repeated string
concatenation
- **Pattern match on** Result and Option: Handle both success and failure cases explic
itly

Performance Notes

- Arrays have O(1) access and O(1) amortized append
- Iterators are lazy and don’t allocate intermediate collections
- StringBuilder is more efficient than string concatenation for building large str
ings
- Physical equality is faster than structural equality but should be used carefull
y