Table of Contents

1 Builtin Package Documentation
1.1 Core Types and Functions
1.1.1 Assertions and Testing
1.1.2 Inspect Function
1.2 Result Type
1.3 Option Type
1.4 Iterator Type
1.5 Array and FixedArray
1.6 String Operations
1.7 StringBuilder
1.8 JSON Support
1.9 Comparison Operations
1.10 Utility Functions
1.11 Error Handling
1.12 Best Practices
1.13 Performance Notes

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:

```

test "assertions" {

  assert_eq(1 + 1, 2)
  assert_eq("hello", "hello")


  assert_true(5 > 3)
  assert_false(2 > 5)


  assert_not_eq(1, 2)
  assert_not_eq("foo", "bar")
}
```

Inspect Function

The inspect function is used for testing and debugging:

```

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

Result Type

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

```
test "result type" {
  fn divide(a : Int, b : Int) -> Result[Int, String] {
    if b == 0 {
      Err("Division by zero")
    } else {
      Ok(a / b)
    }
  }


  let result1 = divide(10, 2)
  inspect(result1, content="Ok(5)")


  let result2 = divide(10, 0)
  inspect(result2, content="Err(\"Division by zero\")")


  match result1 {
    Ok(value) => inspect(value, content="5")
    Err(_) => inspect(false, content="true")
  }
}
```

Option Type

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

```
test "option type" {
  fn find_first_even(numbers : Array[Int]) -> Int? {
    for num in numbers {
      if num % 2 == 0 {
        return Some(num)
      }
    }
    None
  }


  let result1 = find_first_even([1, 3, 4, 5])
  inspect(result1, content="Some(4)")


  let result2 = find_first_even([1, 3, 5])
  inspect(result2, content="None")


  match result1 {
    Some(value) => inspect(value, content="4")
    None => inspect(false, content="true")
  }
}
```

Iterator Type

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

```

test "iterators" {

  let numbers = [1, 2, 3, 4, 5]
  let iter = numbers.iter()


  let collected = iter.collect()
  inspect(collected, content="[1, 2, 3, 4, 5]")


  let doubled = numbers.iter().map(fn(x) { x * 2 }).collect()
  inspect(doubled, content="[2, 4, 6, 8, 10]")


  let evens = numbers.iter().filter(fn(x) { x % 2 == 0 }).collect()
  inspect(evens, content="[2, 4]")


  let sum = numbers.iter().fold(init=0, fn(acc, x) { acc + x })
  inspect(sum, content="15")
}
```

Array and FixedArray

Built-in array types for storing collections:

```

test "arrays" {

  let arr = Array::new()
  arr.push(1)
  arr.push(2)
  arr.push(3)
  inspect(arr, content="[1, 2, 3]")


  let fixed_arr = [10, 20, 30]
  inspect(fixed_arr, content="[10, 20, 30]")


  let length = fixed_arr.length()
  inspect(length, content="3")
  let first = fixed_arr[0]
  inspect(first, content="10")
}
```

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 expected 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 data transformations
- **Use** StringBuilder for string concatenation: More efficient than repeated string concatenation
- **Pattern match on** Result and Option: Handle both success and failure cases explicitly

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 strings
- Physical equality is faster than structural equality but should be used carefully