Table of Contents

Test Package Documentation

- hello
  - moonbit
  - test

- world

This package provides testing utilities and assertion functions for MoonBit programs. 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:

```
test "conditional failure" {
  let value = 10
  if value < 0 {
    @test fail("Value should not be negative: \{value}")
  }


  inspect(value, content="10")
}
```

Test Output and Logging

Create structured test outputs using the Test type:

```

test "test output" {
  let t = @test new("Example Test")


  t write("Testing basic functionality: ")
  t writeln("PASS")


  t writeln("Step 1: Initialize data")
  t writeln("Step 2: Process data")
  t writeln("Step 3: Verify results")


}
```

Snapshot Testing

Compare test outputs against saved snapshots:

```

test "snapshot testing" {
  let t = @test new("Snapshot Test")


  t writeln("Current timestamp: 2024-01-01")
  t writeln("Processing items: [1, 2, 3, 4, 5]")
  t writeln("Result: SUCCESS")



  t snapshot(filename="test_output")
}
```

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:

```

test "property testing" {
  fn is_even(n : Int) -> Bool {
    n % 2 == 0
  }


  let test_values = [0, 2, 4, 6, 8, 10]
  for value in test_values {
    if not(is_even(value)) {
      @test fail("Expected \{value} to be even")
    }
  }


  let odd_values = [1, 3, 5, 7, 9]
  for value in odd_values {
    if is_even(value) {
      @test fail("Expected \{value} to be odd")
    }
  }
}
```

Test Organization

Grouping Related Tests

Use descriptive test names to group related functionality:

```

test "string operations - concatenation" {
  let result = "hello" + " " + "world"
  inspect(result, content="hello world")
}


test "string operations - length" {
  let text = "MoonBit"
  inspect(text length(), content="7")
}


test "string operations - substring" {
  let text = "Hello, World!"
  let sub = text length()
  inspect(sub, content="13")
}
```

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:

```


test "array_push_increases_length" {
  let arr = Array::new()
  let initial_length = arr length()
  arr push(42)
  let new_length = arr length()
  inspect(new_length, content="\{initial_length + 1}")
}



test "array_push_adds_element_at_end" {
  let arr = Array::new()
  arr push(10)
  arr push(20)
  inspect(arr[arr length() - 1], content="20")
}
```

Use Meaningful Test Data

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

```

test "tax_calculation_for_standard_rate" {
  let price = 100
  let tax_rate = 8
  let calculated_tax = price * tax_rate / 100
  inspect(calculated_tax, content="8")
}
```

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.

byte

A package for working with bytes (8-bit unsigned integers) in MoonBit.

Constants

The package provides constants for the minimum and maximum values of a byte:

```

test "byte constants" {
  inspect(@byte min_value, content="b’\\x00’")
  inspect(@byte max_value, content="b’\\xFF’")
}
```

Conversion

Bytes can be converted to other numeric types. The package provides conversion t
o UInt64:

```

test "byte conversion" {
  let byte = b’A’
  inspect(byte to_uint64(), content="65")
  let byte = b’ ’
  inspect(byte to_uint64(), content="32")
}
```

Byte Literals

Although not directly part of this package, MoonBit provides byte literals with the b prefix:

```

test "byte literals" {

  let a = b’a’
  inspect(a to_uint64(), content="97")


  let hex = b’\x41’
  inspect(hex to_uint64(), content="65")


  let null = b’\x00’
  inspect(null to_uint64(), content="0")


  let max = b’\xff’
  inspect(max to_uint64(), content="255")
}
```

Note: The same conversion method can be called either as a method (

```

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

HashMap

A mutable hash map based on a Robin Hood hash table.

Usage

Create

You can create an empty map using new() or construct it using from_array().

```

test {
  let _map2 : @hashmap HashMap[String, Int] = @hashmap new()

}
```

Set & Get

You can use set() to add a key-value pair to the map, and use get() to get a value.

```
test {
  let map : @hashmap HashMap[String, Int] = @hashmap new()
  map set("a", 1)
  assert_eq(map get("a"), Some(1))
  assert_eq(map get_or_default("a", 0), 1)
  assert_eq(map get_or_default("b", 0), 0)
  map remove("a")
  assert_eq(map contains("a"), false)
}
```

Remove

You can use remove() to remove a key-value pair.

```
test {
  let map = @hashmap of([("a", 1), ("b", 2), ("c", 3)])
  map remove("a") |> ignore
  assert_eq(map to_array(), [("c", 3), ("b", 2)])
}
```

Contains

You can use contains() to check whether a key exists.

```
test {
  let map = @hashmap of([("a", 1), ("b", 2), ("c", 3)])
  assert_eq(map contains("a"), true)
  assert_eq(map contains("d"), false)
}
```

Size & Capacity

You can use size() to get the number of key-value pairs in the map, or capacit
y() to get the current capacity.

```
test {
  let map = @hashmap of([("a", 1), ("b", 2), ("c", 3)])
  assert_eq(map size(), 3)
  assert_eq(map capacity(), 8)
}
```

Similarly, you can use is_empty() to check whether the map is empty.

```
test {
  let map : @hashmap HashMap[String, Int] = @hashmap new()
  assert_eq(map is_empty(), true)
}
```

Clear

You can use clear to remove all key-value pairs from the map, but the allocated memory will not change.

```

test {
  let map = @hashmap of([("a", 1), ("b", 2), ("c", 3)])
  map clear()
  assert_eq(map is_empty(), true)
}
```

Iteration

You can use each() or eachi() to iterate through all key-value pairs.

```

test {
  let map = @hashmap of([("a", 1), ("b", 2), ("c", 3)])
  let arr = []
  map each((k, v) => arr push((k, v)))
  let arr2 = []
  map eachi((i, k, v) => arr2 push((i, k, v)))
}
```

Or use iter() to get an iterator of hashmap.

```

test {
  let map = @hashmap of([("a", 1), ("b", 2), ("c", 3)])
  let _iter = map iter()

}
```

String Package Documentation

This package provides comprehensive string manipulation utilities for MoonBit, i
ncluding string creation, conversion, searching, and Unicode handling.

String Creation and Conversion

Create strings from various sources:

```
test "string creation" {

  let chars = ['H', 'e', 'l', 'l', 'o']
  let str1 = String::from_array(chars)
  inspect(str1, content="Hello")


  let str2 = String::from_iter(['W', 'o', 'r', 'l', 'd'] iter())
  inspect(str2, content="World")


  let empty = String::default()
  inspect(empty, content="")
}
```

String Iteration

Iterate over Unicode characters in strings:

```
test "string iteration" {
  let text = "Hello??


  let chars = text iter() collect()
  inspect(chars, content="['H', 'e', 'l', 'l', 'o', '??']"


  let reversed = text rev_iter() collect()
  inspect(reversed, content="['??', 'o', 'l', 'l', 'e', 'H']"


  let mut count = 0
  let mut first_char = 'a'
  text
   iter2()
   each(fn(idx, char) {
    if idx == 0 {
      first_char = char
    }
    count = count + 1
  })
  inspect(first_char, content="H")
  inspect(count, content="6")
}
```

String Conversion

Convert strings to other formats:

```

test "string conversion" {
  let text = "Hello"


  let chars = text to_array()
  inspect(chars, content="[’H’, ’e’, ’l’, ’l’, ’o’]")


  let bytes = text to_bytes()
  inspect(bytes length(), content="10")
}
```

Unicode Handling

Work with Unicode characters and surrogate pairs:

```

test "unicode handling" {
  let emoji_text = "Hello??World


  let char_count = emoji_text iter() count()
  let code_unit_count = emoji_text length()
  inspect(char_count, content="11")
  inspect(code_unit_count, content="12")


  let offset = emoji_text offset_of_nth_char(5)
  inspect(offset, content="Some(5)")


  let has_11_chars = emoji_text char_length_eq(11)
  inspect(has_11_chars, content="true")
}
```

String Comparison

Strings are ordered using shortlex order by Unicode code points:

```

test "string comparison" {
  let result1 = "apple" compare("banana")
  inspect(result1, content="-1")
  let result2 = "hello" compare("hello")
  inspect(result2, content="0")
  let result3 = "zebra" compare("apple")
  inspect(result3, content="1")
}
```

String Views

String views provide efficient substring operations without copying:

```

test "string views" {
  let text = "Hello, World!"
  let view = text[:][7:12]


  let chars = view iter() collect()
  inspect(chars, content="['W', 'o', 'r', 'l', 'd']")


  let substring = view to_string()
  inspect(substring, content="World")
}
```

Practical Examples

Common string manipulation tasks:

```

test "practical examples" {
  let text = "The quick brown fox"


  let words = text split(" ") collect()
  inspect(words length(), content="4")
  inspect(words[0] to_string(), content="The")
  inspect(words[3] to_string(), content="fox")


  let word_strings = words map(fn(v) { v to_string() })
  let mut result = ""
  for i, word in word_strings iter2() {
    if i > 0 {
      result = result + "-"
    }
    result = result + word
  }
  inspect(result, content="The-quick-brown-fox")


  let upper = text[:] to_upper() to_string()
  inspect(upper, content="THE QUICK BROWN FOX")
  let lower = text[:] to_lower() to_string()
  inspect(lower, content="the quick brown fox")
}
```

Performance Notes

- Use StringBuilder or Buffer for building strings incrementally rather than repea
ted concatenation
- String views are lightweight and don’t copy the underlying data
- Unicode iteration handles surrogate pairs correctly but is slower than UTF-16 co
de unit iteration
- Character length operations (char_length_eq, char_length_ge) have O(n) compl
exity where n is the character count

Moonbit/Core Result

Overview

Result[T,E] is a type used for handling computation results and errors in an exp
licit and declarative manner, similar to Rust (Result<T,E>) and OCaml ((’a,
’e) result). It is an enum with two variants: Ok(T), which represents succes
s and contains a value of type T, and Err(E), representing error and containin
g an error value of type E.

Usage

Constructing Result

You can create a Result value using the Ok and Err constructors, remember to giv
e proper type annotations.

```

test {
  let _result : Result[Int, String] = Ok(42)
  let _error : Result[Int, String] = Err("Error message")

}
```

Or use the ok and err functions to create a Result value.

```

test {
  let _result : Result[String, Unit] = Ok("yes")
  let _error : Result[Int, String] = Err("error")

}
```

Querying variant

You can check the variant of a Result using the is_ok and is_err methods.

```

test {
  let result : Result[Int, String] = Ok(42)
  let is_ok = result is Ok(_)
  assert_eq(is_ok, true)
  let is_err = result is Err(_)
  assert_eq(is_err, false)
}
```

Extracting values

You can extract the value from a Result using the match expression (Pattern Matching).

```

test {
  let result : Result[Int, Unit] = Ok(33)
  let val = match result {
    Ok(value) => value
    Err(_) => -1
  }
  assert_eq(val, 33)
}
```

Or using the unwrap method, which will panic if the result is Err and return the value if it is Ok.

```

test {
  let result : Result[Int, String] = Ok(42)
  let value = result unwrap()
  assert_eq(value, 42)
}
```

A safe alternative is the or method, which returns the value if the result is Ok or a default value if it is Err.

```

test {
  let result : Result[Int, String] = Err("error")
  let value = result or(0)
  assert_eq(value, 0)
}
```

There is a lazy version of or called or_else, which takes a function that returns a default value.

```

test {
  let result : Result[Int, String] = Err("error")
  let value = result or_else(() => 0)
  assert_eq(value, 0)
}
```

Transforming values

To transform values inside a Result, you can use the map method, which applies a function to the value if the result is Ok, and remains unchanged if it is Err.

```

test {
  let result : Result[Int, String] = Ok(42)
  let new_result = result map(x => x + 1)
  assert_eq(new_result, Ok(43))
}
```

A dual method to map is map_err, which applies a function to the error value if the result is Err, and remains unchanged if it is Ok.

```
test {
  let result : Result[Int, String] = Err("error")
  let new_result = result.map_err(x => x + "!")
  assert_eq(new_result, Err("error!"))
}
```

You can turn a Result[T, E] into a Option[T] by using the method to_option, whic
h returns Some(value) if the result is Ok, and None if it is Err.

```
test {
  let result : Result[Int, String] = Ok(42)
  let option = result.to_option()
  assert_eq(option, Some(42))
  let result1 : Result[Int, String] = Err("error")
  let option1 = result1.to_option()
  assert_eq(option1, None)
}
```

Monadic operations

Moonbit provides monadic operations for Result, such as flatten and bind, which
allow chaining of computations that return Result.

```
test {
  let result : Result[Result[Int, String], String] = Ok(Ok(42))
  let flattened = result.flatten()
  assert_eq(flattened, Ok(42))
}
```

The bind method is similar to map, but the function passed to it should return a
Result value.

```
test {
  let result : Result[Int, String] = Ok(42)
  let new_result = result.bind(x => Ok(x + 1))
  assert_eq(new_result, Ok(43))
}
```

uint16

The moonbitlang/core/uint16 package provides functionality for working with 16-b
it unsigned integers. This package includes constants, operators, and conversion
s for UInt16 values.

Constants

The package defines the minimum and maximum values for UInt16:

```
test "UInt16 constants" {

  inspect(@uint16 min_value, content="0")


  inspect(@uint16 max_value, content="65535")
}
```

Arithmetic Operations

UInt16 supports standard arithmetic operations:

```
test "UInt16 arithmetic" {
  let a : UInt16 = 100
  let b : UInt16 = 50


  inspect(a + b, content="150")


  inspect(a - b, content="50")


  inspect(a * b, content="5000")


  inspect(a / b, content="2")


  inspect(@uint16 max_value + 1, content="0")
  inspect(@uint16 min_value - 1, content="65535")
}
```

Bitwise Operations

UInt16 supports various bitwise operations:

```
test "UInt16 bitwise operations" {
  let a : UInt16 = 0b1010
  let b : UInt16 = 0b1100


  inspect(a & b, content="8")


  inspect(a | b, content="14")


  inspect(a ^ b, content="6")


  inspect(a << 1, content="20")
  inspect(a << 2, content="40")


  inspect(a >> 1, content="5")
  inspect(b >> 2, content="3")
}
```

Comparison and Equality

UInt16 supports comparison and equality operations:

```
test "UInt16 comparison and equality" {
  let a : UInt16 = 100
  let b : UInt16 = 50
  let c : UInt16 = 100


  inspect(a == c, content="true")
  inspect(a != b, content="true")


  inspect(a > b, content="true")
  inspect(b < a, content="true")
  inspect(a >= c, content="true")
  inspect(c <= a, content="true")
}
```

Default Value and Hashing

UInt16 implements the Default trait:

```

test "UInt16 default value" {

  let a : UInt16 = 0
  inspect(a, content="0")


  let value : UInt16 = 42
  inspect(value hash(), content="42")
}
```

Type Conversions

UInt16 works with various conversions to and from other types:

```

test "UInt16 conversions" {

  inspect((42) to_uint16(), content="42")


  let value : UInt16 = 100
  inspect(value to_int(), content="100")


  inspect((-1) to_uint16(), content="65535")
  inspect((65536) to_uint16(), content="0")
  inspect((65537) to_uint16(), content="1")


  inspect(b’A’ to_uint16(), content="65")
  inspect(b’\xFF’ to_uint16(), content="255")
}
```

Tuple

Tuple is a fixed-size collection of elements of different types. It is a lightweight data structure that can be used to store multiple values in a single variable. This sub-package introduces utils for binary tuples.

Usage

Create

Create a new tuple using the tuple literal syntax.

```

test {
  let tuple2 = (1, 2)
  let tuple3 = (1, 2, 3)
  inspect((tuple2, tuple3), content="((1, 2), (1, 2, 3))")
}
```

Access

You can access the elements of the tuple using pattern match or dot access.

```

test {
  let tuple = (1, 2)
  assert_eq(tuple 0, 1)
  assert_eq(tuple 1, 2)
  let (a, b) = tuple
  assert_eq(a, 1)
  assert_eq(b, 2)
}
```

Transformation

You can transform the tuple using the matrix functions combined with then.

```

test {
  let tuple = (1, 2)
  let tuple2 = ((pair : (Int, Int)) => (pair 0 + 1, pair 1))(tuple)
  inspect(tuple2, content="(2, 2)")
  let tuple3 = tuple |> then(pair => (pair 0, pair 1 + 1))
  inspect(tuple3, content="(1, 3)")
  let mapped = tuple |> then(pair => (pair 0 + 1, pair 1 - 1))
  inspect(mapped, content="(2, 1)")
}
```

List

The List package provides an immutable linked list data structure with a variety
 of utility functions for functional programming.

Table of Contents

Overview

List is a functional, immutable data structure that supports efficient traversal
, transformation, and manipulation. It is particularly useful for recursive algo
rithms and scenarios where immutability is required.

Performance

- **prepend**: O(1)
- **length**: O(n)
- **map/filter**: O(n)
- **concatenate**: O(n)
- **reverse**: O(n)
- **nth**: O(n)
- **sort**: O(n log n)
- **flatten**: O(n * m), where m is the average inner list length
- **space complexity**: O(n)

Usage

Create

You can create an empty list or a list from an array.

```
test {
  let empty_list : @list List[Int] = @list new()
  assert_true(empty_list is_empty())
  let list = @list of([1, 2, 3, 4, 5])
  assert_eq(list, @list of([1, 2, 3, 4, 5]))
}
```

Basic Operations

Prepend

Add an element to the beginning of the list.

```
test {
  let list = @list of([2, 3, 4, 5]) prepend(1)
  assert_eq(list, @list of([1, 2, 3, 4, 5]))
}
```

Length

Get the number of elements in the list.

```
test {
  let list = @list of([1, 2, 3, 4, 5])
  assert_eq(list length(), 5)
}
```

Check if Empty

Determine if the list is empty.

```
test {
  let empty_list : @list List[Int] = @list new()
  assert_eq(empty_list is_empty(), true)
}
```

Access Elements

Head

Get the first element of the list as an Option.

```

test {
  let list = @list of([1, 2, 3, 4, 5])
  assert_eq(list head(), Some(1))
}
```

Tail

Get the list without its first element.

```

test {
  let list = @list of([1, 2, 3, 4, 5])
  assert_eq(list unsafe_tail(), @list of([2, 3, 4, 5]))
}
```

Nth Element

Get the nth element of the list as an Option.

```

test {
  let list = @list of([1, 2, 3, 4, 5])
  assert_eq(list nth(2), Some(3))
}
```

Iteration

Each

Iterate over the elements of the list.

```

test {
  let arr = []
  @list of([1, 2, 3, 4, 5]) each(x => arr push(x))
  assert_eq(arr, [1, 2, 3, 4, 5])
}
```

Map

Transform each element of the list.

```

test {
  let list = @list of([1, 2, 3, 4, 5]) map(x => x * 2)
  assert_eq(list, @list of([2, 4, 6, 8, 10]))
}
```

Filter

Keep elements that satisfy a predicate.

```
test {
  let list = @list of([1, 2, 3, 4, 5]) filter(x => x % 2 == 0)
  assert_eq(list, @list of([2, 4]))
}
```

Advanced Operations

Reverse

Reverse the list.

```
test {
  let list = @list of([1, 2, 3, 4, 5]) rev()
  assert_eq(list, @list of([5, 4, 3, 2, 1]))
}
```

Concatenate

Concatenate two lists.

```
test {
  let list = @list of([1, 2, 3]) concat(@list of([4, 5]))
  assert_eq(list, @list of([1, 2, 3, 4, 5]))
}
```

Flatten

Flatten a list of lists.

```
test {
  let list = @list of([@list of([1, 2]), @list of([3, 4])]) flatten()
  assert_eq(list, @list of([1, 2, 3, 4]))
}
```

Sort

Sort the list in ascending order.

```
test {
  let list = @list of([3, 1, 4, 1, 5, 9]) sort()
  assert_eq(list, @list of([1, 1, 3, 4, 5, 9]))
}
```

Conversion

To Array

Convert a list to an array.

```

test {
  let list = @list of([1, 2, 3, 4, 5])
  assert_eq(list to_array(), [1, 2, 3, 4, 5])
}
```

From Array

Create a list from an array.

```

test {
  let list = @list from_array([1, 2, 3, 4, 5])
  assert_eq(list, @list of([1, 2, 3, 4, 5]))
}
```

Equality

Lists with the same elements in the same order are considered equal.

```

test {
  let list1 = @list of([1, 2, 3])
  let list2 = @list of([1, 2, 3])
  assert_eq(list1 == list2, true)
}
```

Error Handling Best Practices

When accessing elements that might not exist, use pattern matching for safety:

```

fn safe_head(list : @list List[Int]) -> Int {
  match list head() {
    Some(value) => value
    None => 0
  }
}


test {
  let list = @list of([1, 2, 3])
  assert_eq(safe_head(list), 1)
  let empty_list : @list List[Int] = @list new()
  assert_eq(safe_head(empty_list), 0)
}
```

Additional Error Cases

- **nth()** on an empty list or out-of-bounds index: Returns None.
- **tail()** on an empty list: Returns Empty.
- **sort()** with non-comparable elements: Throws a runtime error.

Implementation Notes

The List is implemented as a singly linked list. Operations like prepend and hea
d are O(1), while operations like length and map are O(n).

Key properties of the implementation:
- Immutable by design
- Recursive-friendly
- Optimized for functional programming patterns

Comparison with Other Collections

- **@array.T**: Provides O(1) random access but is mutable; use when random access i
s required.
- **@list.T**: Immutable and optimized for recursive operations; use when immutability
 and functional patterns are required.

Choose List when you need:
- Immutable data structures
- Efficient prepend operations
- Functional programming patterns

Priority Queue

A priority queue is a data structure capable of maintaining maximum/minimum valu
es at front of the queue, which may have other names in other programming langua
ges (C++ std::priority_queue / Rust BinaryHeap). The priority queue here is i
mplemented as a pairing heap and has excellent performance.

Usage

Create

You can use new() or of() to create a priority queue.

```
test {
  let queue1 : @priority_queue T[Int] = @priority_queue new()
  let queue2 = @priority_queue of([1, 2, 3])
  @json inspect(queue1, content=[])
  @json inspect(queue2, content=[3, 2, 1])
}
```

Note, however, that the default priority queue created is greater-first; if you
need to create a less-first queue, you can write a struct belongs to Compare tra
it to implement it.

Length

You can use length() to get the number of elements in the current priority que
ue.

```
test {
  let pq = @priority_queue of([1, 2, 3, 4, 5])
  assert_eq(pq length(), 5)
}
```

Similarly, you can use the is_empty to determine whether the priority queue is e
mpty.

```
test {
  let pq : @priority_queue T[Int] = @priority_queue new()
  assert_eq(pq is_empty(), true)
}
```

Peek

You can use peek() to look at the head element of a queue, which must be eithe
r the maximum or minimum value of an element in the queue, depending on the natu
re of the specification. The return value of peek() is an Option, which means
that the result will be None when the queue is empty.

```
test {
  let pq = @priority_queue of([1, 2, 3, 4, 5])
  assert_eq(pq peek(), Some(5))
}
```

Push

You can use push() to add elements to the priority queue.

```
test {
  let pq : @priority_queue T[Int] = @priority_queue new()
  pq push(1)
  pq push(2)
  assert_eq(pq peek(), Some(2))
}
```

Pop

You can use pop() to pop the element at the front of the priority queue, respectively, and like Peek, its return values are Option, loaded with the value of the element being popped.

```
test {
  let pq = @priority_queue of([5, 4, 3, 2, 1])
  assert_eq(pq pop(), Some(5))
}
```

```
test {
  let pq = @priority_queue of([5, 4, 3, 2, 1])
  assert_eq(pq length(), 5)
}
```

Clear

You can use clear to clear a priority queue.

```
test {
  let pq = @priority_queue of([1, 2, 3, 4, 5])
  pq clear()
  assert_eq(pq is_empty(), true)
}
```

Copy and Transfer

You can copy a priority queue using the copy method.

```
test {
  let pq = @priority_queue of([1, 2, 3])
  let _pq2 = pq copy()

}
```

Array Package Documentation

This package provides array manipulation utilities for MoonBit, including fixed-size arrays (FixedArray), dynamic arrays (Array), and array views (ArrayView/View).

Creating Arrays

There are several ways to create arrays in MoonBit:

```
test "array creation" {

  let arr1 = [1, 2, 3]
  inspect(arr1, content="[1, 2, 3]")


  let arr2 = Array::makei(3, i => i * 2)
  inspect(arr2, content="[0, 2, 4]")


  let arr3 = Array::from_iter("hello" iter())
  inspect(arr3, content="['h', 'e', 'l', 'l', 'o']")
}
```

Array Operations

Common array operations include mapping, filtering, and folding:

```
test "array operations" {
  let nums = [1, 2, 3, 4, 5]


  let neg_evens = nums filter_map(x => if x % 2 == 0 { Some(-x) } else {
  inspect(neg_evens, content="[-2, -4]")


  let sum = nums fold(init=0, (acc, x) => acc + x)
  inspect(sum, content="15")


  let last = nums last()
  inspect(last, content="Some(5)")
}
```

Sorting

The package provides various sorting utilities:

```

test "sorting" {
  let arr = [3, 1, 4, 1, 5, 9, 2, 6]


  let sorted1 = arr copy()
  sorted1 sort()
  inspect(sorted1, content="[1, 1, 2, 3, 4, 5, 6, 9]")


  let strs = ["aa", "b", "ccc"]
  let sorted2 = strs copy()
  sorted2 sort_by((a, b) => a length() compare(b length()))
  inspect(
    sorted2,
    content=(
      #|["b", "aa", "ccc"]
    ),
  )


  let pairs = [(2, "b"), (1, "a"), (3, "c")]
  let sorted3 = pairs copy()
  sorted3 sort_by_key(p => p 0)
  inspect(
    sorted3,
    content=(
      #|[(1, "a"), (2, "b"), (3, "c")]
    ),
  )
}
```

Array Views

Array views provide a lightweight way to work with array slices:

```

test "array views" {
  let arr = [1, 2, 3, 4, 5]
  let view = arr[1:4]
  inspect(view, content="[2, 3, 4]")


  let doubled = view map(x => x * 2)
  inspect(doubled, content="[4, 6, 8]")
}
```

Fixed Arrays

Fixed arrays provide immutable array operations:

```

test "fixed arrays" {
  let fixed : FixedArray[_] = [1, 2, 3]


  let combined = fixed + [4, 5]
  inspect(combined, content="[1, 2, 3, 4, 5]")


  let has_two = fixed contains(2)
  inspect(has_two, content="true")


  let starts = fixed starts_with([1, 2])
  inspect(starts, content="true")
  let ends = fixed ends_with([2, 3])
  inspect(ends, content="true")
}
```

Utilities

Additional array utilities for common operations:

```

test "utilities" {

  let words = ["hello", "world"]
  let joined = words join(" ")
  inspect(joined, content="hello world")


  let nums = [1, 2, 3, 4, 5]


  let shuffled = nums shuffle(rand=_ => 1)
  inspect(shuffled, content="[1, 3, 4, 5, 2]")
}
```

Sorted Set

A mutable set backed by a red-black tree.

Usage

Create

You can create an empty SortedSet or a SortedSet from other containers.

```

test {
  let _set1 : @sorted_set SortedSet[Int] = @sorted_set new()
  let _set2 = @sorted_set singleton(1)
  let _set3 = @sorted_set from_array([1])

}
```

Container Operations

Add an element to the SortedSet in place.

```

test {
  let set4 = @sorted_set from_array([1, 2, 3, 4])
  set4 add(5)
  let set6 = @sorted_set from_array([1, 2, 3, 4, 5])
  assert_eq(set6 to_array(), [1, 2, 3, 4, 5])
}
```

Remove an element from the SortedSet in place.

```

test {
  let set = @sorted_set from_array([3, 8, 1])
  set remove(8)
  let set7 = @sorted_set from_array([1, 3])
  assert_eq(set7 to_array(), [1, 3])
}
```

Whether an element is in the set.

```

test {
  let set = @sorted_set from_array([1, 2, 3, 4])
  assert_eq(set contains(1), true)
  assert_eq(set contains(5), false)
}
```

Iterates over the elements in the set.

```

test {
  let arr = []
  @sorted_set from_array([1, 2, 3, 4]) each(v => arr push(v))
  assert_eq(arr, [1, 2, 3, 4])
}
```

Get the size of the set.

```

test {
  let set = @sorted_set from_array([1, 2, 3, 4])
  assert_eq(set size(), 4)
}
```

Whether the set is empty.

```
test {
  let set : @sorted_set SortedSet[Int] = @sorted_set new()
  assert_eq(set is_empty(), true)
}
```

Set Operations

Union, intersection and difference of two sets. They return a new set that does not overlap with the original sets in memory.

```
test {
  let set1 = @sorted_set from_array([3, 4, 5])
  let set2 = @sorted_set from_array([4, 5, 6])
  let set3 = set1 union(set2)
  assert_eq(set3 to_array(), [3, 4, 5, 6])
  let set4 = set1 intersection(set2)
  assert_eq(set4 to_array(), [4, 5])
  let set5 = set1 difference(set2)
  assert_eq(set5 to_array(), [3])
}
```

Determine the inclusion and separation relationship between two sets.

```
test {
  let set1 = @sorted_set from_array([1, 2, 3])
  let set2 = @sorted_set from_array([7, 2, 9, 4, 5, 6, 3, 8, 1])
  assert_eq(set1 subset(set2), true)
  let set3 = @sorted_set from_array([4, 5, 6])
  assert_eq(set1 disjoint(set3), true)
}
```

Stringify

SortedSet implements to_string (i.e. Show trait), which allows you to directly output it.

```
test {
  let set = @sorted_set from_array([1, 2, 3])
  assert_eq(set to_string(), "@sorted_set.from_array([1, 2, 3])")
}
```

int

The moonbitlang/core/int package provides essential operations on 32-bit integers.

Basic Operations

This section shows the basic operations available for integers:

```

test "basic int operations" {

  inspect(@int abs(-42), content="42")
  inspect(@int abs(42), content="42")


  inspect(@int min_value, content="-2147483648")
  inspect(@int max_value, content="2147483647")
}
```

Byte Conversion

The package provides methods to convert integers to their byte representation in both big-endian and little-endian formats:

```

test "byte conversions" {
  let num = 258


  let be_bytes = num to_be_bytes()
  inspect(
    be_bytes to_string(),
    content=(
      #|b"\x00\x00\x01\x02"
    ),
  )


  let le_bytes = num to_le_bytes()
  inspect(
    le_bytes to_string(),
    content=(
      #|b"\x02\x01\x00\x00"
    ),
  )
}
```

Method Syntax

All operations are also available using method syntax for better readability:

```

test "method syntax" {
  let n = -42


  inspect(n abs(), content="42")


  let be = n to_be_bytes()
  let le = n to_le_bytes()
  inspect(
    be to_string(),
    content=(
      #|b"\xff\xff\xff\xd6"
    ),
  )
  inspect(
    le to_string(),
    content=(
      #|b"\xd6\xff\xff\xff"
    ),
  )
}
```

The package provides the foundations for 32-bit integer operations in MoonBit, essential for any numeric computation.

int64

This package provides operations for working with 64-bit signed integers (Int64) in MoonBit.

Basic Operations

Int64 values can be created from regular 32-bit integers using from_int. The package also provides constants for the maximum and minimum values representable by Int64.

```

test "basic operations" {
  let i : Int64 = -12345L

  inspect(@int64 from_int(-12345) == i, content="true")


  inspect(@int64 max_value, content="9223372036854775807")
  inspect(@int64 min_value, content="-9223372036854775808")


  inspect(@int64 abs(i), content="12345")
}
```

Binary Representation

The package provides functions to convert Int64 values to their binary represent ation in both big-endian and little-endian byte order:

```

test "binary conversion" {
  let x = 258L
  let be_bytes = x to_be_bytes()
  let le_bytes = x to_le_bytes()


  inspect(
    be_bytes to_string(),
    content=(
      #|b"\x00\x00\x00\x00\x00\x00\x01\x02"
    ),
  )
  inspect(
    le_bytes to_string(),
    content=(
      #|b"\x02\x01\x00\x00\x00\x00\x00\x00"
    ),
  )


  let len = be_bytes length()
  inspect(len, content="8")
}
```

Method-Style Usage

All operations are also available as methods on Int64 values:

```

test "method style" {
  let x = -42L


  inspect(x abs(), content="42")


  inspect(
    x to_be_bytes(),
    content=(
      #|b"\xff\xff\xff\xff\xff\xff\xff\xd6"
    ),
  )
}
```

Note that Int64 implements the Hash trait, allowing it to be used as keys in has h maps and members of hash sets.

unit

The unit package provides functionality for working with the singleton type Unit, which represents computations that produce side effects but return no meaningful value. This is a fundamental type in functional programming for operations like I/O, logging, and state modifications.

Understanding Unit Type

The Unit type has exactly one value: (). This might seem trivial, but it serves important purposes in type systems:

- **Side Effect Indication**: Functions returning Unit signal they’re called for side effects
- **Placeholder Type**: Used when a type parameter is needed but no meaningful value exists
- **Functional Programming**: Represents "no useful return value" without using null or exceptions
- **Interface Consistency**: Maintains uniform function signatures in generic contexts

Unit Value Creation

The unit value can be created in multiple ways:

```

test "unit construction" {

  let u1 = ()


  let u2 = @unit default()
  fn println(_ : String) {

  }

  inspect(u1 == u2, content="true")


  fn log_message(msg : String) -> Unit {

    println(msg)
    ()
  }

  let result = log_message("Hello, world!")
  inspect(result, content="()")
}
```

Working with Side-Effect Functions

Functions that return Unit are typically called for their side effects:

```

test "side effect patterns" {
  let numbers = [1, 2, 3, 4, 5]
  fn println(_ : Int) {

  }

  let processing_result = numbers fold(init=(), fn(_acc, n) {

    if n % 2 == 0 {

      println(n)
    }
    ()
  })
  inspect(processing_result, content="()")


  numbers each(fn(n) { if n % 2 == 0 { println(n) } })
}
```

String Representation and Debugging

Unit values have a standard string representation for debugging:

```

test "unit string conversion" {
  let u = ()
  inspect(u to_string(), content="()")


  fn perform_operation() -> Unit {

    ()
  }

  let result = perform_operation()
  let debug_msg = "Operation completed: \{result}"
  inspect(debug_msg, content="Operation completed: ()")
}
```

Generic Programming with Unit

Unit is particularly useful in generic contexts where you need to represent "no meaningful value":

```

test "generic unit usage" {

  let items = [1, 2, 3, 4, 5]


  items each(fn(x) {

    let processed = x * 2
    assert_true(processed > 0)
  })


  let completion_status = ()
  inspect(completion_status, content="()")


  let operation_result : Result[Unit, String] = Ok(())
  inspect(operation_result, content="Ok(())")
}
```

Built-in Trait Implementations

Unit implements essential traits for seamless integration with MoonBit’s type sy
stem:

```

test "unit trait implementations" {
  let u1 = ()
  let u2 = ()


  inspect(u1 == u2, content="true")


  inspect(u1 compare(u2), content="0")


  let h1 = u1 hash()
  let h2 = u2 hash()
  inspect(h1 == h2, content="true")


  let u3 = Unit::default()
  inspect(u3 == u1, content="true")
}
```

Practical Use Cases

Result Accumulation

```

test "result accumulation" {

  let operations = [
    fn() { () },
    fn() { () },
    fn() { () },
  ]
  let final_result = operations fold(init=(), fn(acc, operation) {
    operation()
    acc
  })
  inspect(final_result, content="()")
}
```

Builder Pattern Termination

```

test "builder pattern" {

  let settings = ["debug=true", "timeout=30"]


  fn apply_config(config_list : Array[String]) -> Unit {

    let _has_settings = config_list length() > 0
    ()
  }

  let result = apply_config(settings)
  inspect(result, content="()")
}
```

The Unit type provides essential functionality for representing "no meaningful r
eturn value" in a type-safe way, enabling clean functional programming patterns
and consistent interfaces across MoonBit code.

BigInt Package Documentation

This package provides arbitrary-precision integer arithmetic through the BigInt
type. BigInt allows you to work with integers of unlimited size, making it perfe
ct for cryptographic operations, mathematical computations, and any scenario whe
re standard integer types are insufficient.

Creating BigInt Values

There are several ways to create BigInt values:

```

test "creating bigint values" {

  let big1 = 12345678901234567890N
  inspect(big1, content="12345678901234567890")


  let big2 = @bigint BigInt::from_int(42)
  inspect(big2, content="42")


  let big3 = @bigint BigInt::from_int64(9223372036854775807L)
  inspect(big3, content="9223372036854775807")


  let big4 = @bigint BigInt::from_string("123456789012345678901234567890
  inspect(big4, content="123456789012345678901234567890")


  let big5 = @bigint BigInt::from_hex("1a2b3c4d5e6f")
  inspect(big5, content="28772997619311")
}
```

Basic Arithmetic Operations

BigInt supports all standard arithmetic operations:

```

test "arithmetic operations" {
  let a = 123456789012345678901234567890N
  let b = 987654321098765432109876543210N


  let sum = a + b
  inspect(sum, content="1111111110111111111011111111100")


  let diff = b - a
  inspect(diff, content="864197532086419753208641975320")


  let product = @bigint BigInt::from_int(123) * @bigint BigInt::from_int
  inspect(product, content="56088")


  let quotient = @bigint BigInt::from_int(1000) / @bigint BigInt::from_i
  inspect(quotient, content="142")


  let remainder = @bigint BigInt::from_int(1000) % @bigint BigInt::from_
  inspect(remainder, content="6")


  let neg = -a
  inspect(neg, content="-123456789012345678901234567890")
}
```

Comparison Operations

Compare BigInt values with each other and with regular integers:

```

test "comparisons" {
  let big = 12345N
  let small = 123N


  inspect(big > small, content="true")
  inspect(big == small, content="false")
  inspect(small < big, content="true")


  inspect(big equal_int(12345), content="true")
  inspect(big compare_int(12345), content="0")
  inspect(big compare_int(1000), content="1")
  inspect(small compare_int(200), content="-1")


  let big64 = @bigint BigInt::from_int64(9223372036854775807L)
  inspect(big64 equal_int64(9223372036854775807L), content="true")
}
```

Bitwise Operations

BigInt supports bitwise operations for bit manipulation:

```

test "bitwise operations" {
  let a = 0b11110000N
  let b = 0b10101010N


  let and_result = a & b
  inspect(and_result, content="160")


  let or_result = a | b
  inspect(or_result, content="250")


  let xor_result = a ^ b
  inspect(xor_result, content="90")


  let big_num = 255N
  inspect(big_num bit_length(), content="8")


  let with_zeros = 1000N
  let ctz = with_zeros ctz()
  inspect(ctz >= 0, content="true")
}
```

Power and Modular Arithmetic

BigInt provides efficient power and modular exponentiation:

```

test "power operations" {

  let base = 2N
  let exponent = 10N
  let power = base pow(exponent)
  inspect(power, content="1024")


  let base2 = 3N
  let exp2 = 5N
  let modulus = 7N
  let mod_power = base2 pow(exp2, modulus~)
  inspect(mod_power, content="5")


  let large_base = 123N
  let large_exp = 20N
  let large_mod = 1000007N
  let result = large_base pow(large_exp, modulus=large_mod)
  inspect(result, content="378446")
}
```

String and Hexadecimal Conversion

Convert BigInt to and from various string representations:

```

test "string conversions" {
  let big = 255N


  let decimal = big to_string()
  inspect(decimal, content="255")


  let hex_lower = big to_hex()
  inspect(hex_lower, content="FF")


  let hex_upper = big to_hex(uppercase=true)
  inspect(hex_upper, content="FF")


  let from_hex = @bigint BigInt::from_hex("deadbeef")
  inspect(from_hex, content="3735928559")


  let original = 98765432109876543210N
  let as_string = original to_string()
  let parsed_back = @bigint BigInt::from_string(as_string)
  inspect(original == parsed_back, content="true")
}
```

Byte Array Conversion

Convert BigInt to and from byte arrays:

```

test "byte conversions" {
  let big = 0x123456789abcdefN


  let bytes = big to_octets()
  inspect(bytes length() > 0, content="true")


  let from_bytes = @bigint BigInt::from_octets(bytes)
  inspect(from_bytes == big, content="true")


  let fixed_length = @bigint BigInt::from_int(255) to_octets(length=4)
  inspect(fixed_length length(), content="4")







}
```

Type Conversions

Convert BigInt to standard integer types:

```

test "type conversions" {
  let big = 12345N


  let as_int = big to_int()
  inspect(as_int, content="12345")


  let as_int64 = big to_int64()
  inspect(as_int64, content="12345")


  let as_uint = big to_uint()
  inspect(as_uint, content="12345")


  let small = 255N
  let as_int16 = small to_int16()
  inspect(as_int16, content="255")
  let as_uint16 = small to_uint16()
  inspect(as_uint16, content="255")
}
```

JSON Serialization

BigInt values can be serialized to and from JSON:

```

test "json serialization" {
  let big = 12345678901234567890N


  let json = big to_json()
  inspect(json, content="String(\"12345678901234567890\")")


  let very_big = @bigint BigInt::from_string("12345678901234567890123456
  let big_json = very_big to_json()
  inspect(big_json, content="String(\"123456789012345678901234567890\")"
}
```

Utility Functions

Check properties of BigInt values:

```

test "utility functions" {
  let zero = 0N
  let positive = 42N
  let negative = -42N


  inspect(zero is_zero(), content="true")
  inspect(positive is_zero(), content="false")


  inspect(positive > zero, content="true")
  inspect(negative < zero, content="true")
  inspect(zero == zero, content="true")
}
```

Use Cases and Applications

BigInt is particularly useful for:

- **Cryptography**: RSA encryption, digital signatures, and key generation
- **Mathematical computations**: Factorial calculations, Fibonacci sequences, prime nu mber testing
- **Financial calculations**: High-precision monetary computations
- **Scientific computing**: Large integer calculations in physics and chemistry
- **Data processing**: Handling large numeric IDs and checksums

Performance Considerations

- BigInt operations are slower than regular integer operations due to arbitrary pr ecision
- Addition and subtraction are generally fast
- Multiplication and division become slower with larger numbers
- Modular exponentiation is optimized for cryptographic use cases
- String conversions can be expensive for very large numbers

Best Practices

- **Use regular integers when possible**: Only use BigInt when you need arbitrary prec ision
- **Cache string representations**: If you need to display the same BigInt multiple ti mes
- **Use modular arithmetic**: For cryptographic applications, always use modular expon entiation
- **Be careful with conversions**: Converting very large BigInt to regular integers wi ll truncate
- **Consider memory usage**: Very large BigInt values consume more memory

Bench Package Documentation

This package provides benchmarking utilities for measuring the performance of Mo
onBit code. It includes functions for timing code execution, collecting statisti
cs, and generating performance reports.

Basic Benchmarking

Use the single_bench function to benchmark individual operations:

```

#skip("slow tests")
test "basic benchmarking" {
  fn simple_calc(n : Int) -> Int {
    n * 2 + 1
  }

  let summary = @bench single_bench(name="simple_calc", fn() {
    ignore(simple_calc(5))
  })


  inspect(summary to_json() stringify() length() > 0, content="true")
}
```

Benchmark Collection

Use the T type to collect multiple benchmarks:

```
#skip("slow tests")
test "benchmark collection" {
  let bencher = @bench new()


  bencher bench(name="array_creation", fn() {
    let arr = Array::new()
    for i in 0..<5 {
      arr push(i)
    }
  })
  bencher bench(name="array_iteration", fn() {
    let arr = [1, 2, 3, 4, 5]
    let mut sum = 0
    for x in arr {
      sum = sum + x
    }
  })


  let report = bencher dump_summaries()
  inspect(report length() > 0, content="true")
}
```

Benchmarking Different Algorithms

Compare the performance of different implementations:

```

#skip("slow tests")
test "algorithm comparison" {
  let bencher = @bench new()


  bencher bench(name="linear_search", fn() {
    let arr = [1, 2, 3, 4, 5]
    let target = 3
    let mut found = false
    for x in arr {
      if x == target {
        found = true
        break
      }
    }
    ignore(found)
  })


  bencher bench(name="builtin_contains", fn() {
    let arr = [1, 2, 3, 4, 5]
    ignore(arr contains(3))
  })
  let results = bencher dump_summaries()
  inspect(results length() > 10, content="true")
}
```

Data Structure Benchmarks

Benchmark different data structure operations:

```

#skip("slow tests")
test "data structure benchmarks" {
  let bencher = @bench.new()


  bencher.bench(name="array_append", fn() {
    let arr = Array::new()
    for i in 0..<5 {
      arr.push(i)
    }
  })


  bencher.bench(name="fixedarray_access", fn() {
    let arr = [0, 1, 2, 3, 4]
    let mut sum = 0
    for i in 0..<arr.length() {
      sum = sum + arr[i]
    }
    ignore(sum)
  })
  let report = bencher.dump_summaries()
  inspect(report.length() > 50, content="true")
}
```

String Operations Benchmarking

Measure string manipulation performance:

```

#skip("slow tests")
test "string benchmarks" {
  let bencher = @bench.new()


  bencher.bench(name="string_concat", fn() {
    let mut result = ""
    for i in 0..<5 {
      result = result + "x"
    }
  })


  bencher.bench(name="stringbuilder", fn() {
    let builder = StringBuilder::new()
    for i in 0..<5 {
      builder.write_string("x")
    }
    ignore(builder.to_string())
  })
  let results = bencher.dump_summaries()
  inspect(results.length() > 50, content="true")
}
```

Memory Usage Prevention

Use keep to prevent compiler optimizations from eliminating benchmarked code:

```

#skip("slow tests")
test "preventing optimization" {
  let bencher = @bench new()
  bencher bench(name="with_keep", fn() {
    let result = Array::makei(5, fn(i) { i * i })

    bencher keep(result)
  })
  let report = bencher dump_summaries()
  inspect(report length() > 30, content="true")
}
```

Iteration Count Control

Control the number of benchmark iterations:

```

#skip("slow tests")
test "iteration control" {
  let bencher = @bench new()


  bencher bench(
    name="stable_benchmark",
    fn() {
      let arr = [1, 2, 3, 4, 5]
      let sum = arr fold(init=0, fn(acc, x) { acc + x })
      ignore(sum)
    },
    count=20,
  )


  bencher bench(
    name="quick_benchmark",
    fn() {
      let mut result = 0
      for i in 0..<10 {
        result = result + i
      }
      ignore(result)
    },
    count=2,
  )
  let results = bencher dump_summaries()
  inspect(results length() > 50, content="true")
}
```

Benchmarking Best Practices

1. Isolate What You’re Measuring

```

#skip("slow tests")
test "isolation example" {
  let bencher = @bench new()


  let data = Array::makei(10, fn(i) { i })
  bencher bench(name="array_sum", fn() {
    let mut sum = 0
    for x in data {
      sum = sum + x
    }
    bencher keep(sum)
  })
  let results = bencher dump_summaries()
  inspect(results length() > 0, content="true")
}
```

2. Warm Up Before Measuring

```

#skip("slow tests")
test "warmup example" {
  let bencher = @bench new()
  fn expensive_operation() -> Int {
    let mut result = 0
    for i in 0..<5 {
      result = result + i * i
    }
    result
  }


  for _ in 0..<5 {
    ignore(expensive_operation())
  }


  bencher bench(name="warmed_up", fn() {
    let result = expensive_operation()
    bencher keep(result)
  })
  let report = bencher dump_summaries()
  inspect(report length() > 30, content="true")
}
```

3. Use Meaningful Names

```

#skip("slow tests")
test "meaningful names" {
  let bencher = @bench.new()


  bencher.bench(name="array_insert_10_items", fn() {
    let arr = Array::new()
    for i in 0..<10 {
      arr.push(i * 2)
    }
    bencher.keep(arr)
  })
  bencher.bench(name="array_search_sorted_10", fn() {
    let arr = Array::makei(10, fn(i) { i })
    let result = arr.contains(5)
    bencher.keep(result)
  })
  let results = bencher.dump_summaries()
  inspect(results.length() > 50, content="true")
}
```

Performance Analysis

The benchmark results include statistical information:

- **Timing measurements**: Microsecond precision timing
- **Statistical analysis**: Median, percentiles, and outlier detection
- **Batch sizing**: Automatic adjustment for stable measurements
- **JSON output**: Machine-readable results for analysis

Integration with Testing

Benchmarks can be integrated into your testing workflow:

```

#skip("slow tests")
test "performance regression test" {
  let bencher = @bench new()


  bencher bench(name="critical_algorithm", fn() {
    let data = [5, 2, 8, 1, 9, 3, 7, 4, 6]
    let sorted = Array::new()
    for x in data {
      sorted push(x)
    }
    sorted sort()
    bencher keep(sorted)
  })
  let results = bencher dump_summaries()

  inspect(results length() > 50, content="true")
}
```

Common Benchmarking Patterns

- **Before/After comparisons**: Benchmark code before and after optimizations
- **Algorithm comparison**: Compare different implementations of the same functionality
- **Scaling analysis**: Benchmark with different input sizes
- **Memory vs. speed tradeoffs**: Compare memory-efficient vs. speed-optimized approaches
- **Platform differences**: Compare performance across different targets (JS, WASM, native)

Tips for Accurate Benchmarks

- Run benchmarks multiple times and look for consistency
- Be aware of system load and other processes affecting timing
- Use appropriate iteration counts (more for stable results, fewer for quick feedback)
- Measure what matters to your use case
- Consider both average case and worst case performance
- Profile memory usage separately if memory performance is important

The bench package provides essential tools for performance analysis and optimization in MoonBit applications.

cmp

This package provides utility functions for comparing values.

Generic Comparison Functions

The library provides generic comparison functions that work with any type implementing the Compare trait:

```

test "generic comparison" {

  inspect(@cmp maximum(3, 4), content="4")
  inspect(@cmp minimum(3, 4), content="3")
}
```

Comparison by Key

With @cmp.maximum_by_key() and @cmp.minimum_by_key(), it is possible to compare values based on arbitrary keys derived from the them. This is particularly useful when you need to compare complex objects based on some specific aspect or field.

```

test "cmp_by_key" {
  struct Person {
    name : String
    age : Int
  } derive(Show)


  let s1 = "hello"
  let s2 = "hi"
  let longer = @cmp maximum_by_key(s1, s2, String::length)
  inspect(longer, content="hello")


  let alice = { name: "Alice", age: 25 }
  let bob = { name: "Bob", age: 30 }
  let younger = @cmp minimum_by_key(alice, bob, p => p age)
  inspect(younger, content="{name: \"Alice\", age: 25}")


  let p1 = ("first", 1)
  let p2 = ("second", 1)
  let snd = (p : (_, _)) => p 1
  assert_eq(@cmp minimum_by_key(p1, p2, snd), p1)
  assert_eq(@cmp maximum_by_key(p1, p2, snd), p2)
}
```

math

This library provides common mathematical functions for floating-point arithmetic, trigonometry, and general numeric comparisons.

Constants

MoonBit math library provides the mathematical constant À:

```

test "mathematical constants" {
  inspect(@math PI, content="3.141592653589793")
}
```

Basic Arithmetic Functions

Rounding Functions

Several functions are available for rounding numbers in different ways:

```

test "rounding functions" {

  inspect(@math round(3.7), content="4")
  inspect(@math round(-3.7), content="-4")


  inspect(@math ceil(3.2), content="4")
  inspect(@math ceil(-3.2), content="-3")


  inspect(@math floor(3.7), content="3")
  inspect(@math floor(-3.7), content="-4")


  inspect(@math trunc(3.7), content="3")
  inspect(@math trunc(-3.7), content="-3")
}
```

Exponential and Logarithmic Functions

The library provides standard exponential and logarithmic operations:

```

test "exponential and logarithmic" {

  inspect(@math exp(1.0), content="2.718281828459045")
  inspect(@math expm1(1.0), content="1.718281828459045")


  inspect(@math ln(2.718281828459045), content="1")
  inspect(@math ln_1p(1.718281828459045), content="1")


  inspect(@math log2(8.0), content="3")
  inspect(@math log10(100.0), content="2")
}
```

Trigonometric Functions

Basic Trigonometric Functions

Standard trigonometric functions operating in radians:

```

test "basic trigonometry" {

  inspect(@math sin(@math PI / 2.0), content="1")
  inspect(@math cos(0.0), content="1")
  inspect(@math tan(@math PI / 4.0), content="0.9999999999999999")


  inspect(@math asin(1.0), content="1.5707963267948966")
  inspect(@math acos(1.0), content="0")
  inspect(@math atan(1.0), content="0.7853981633974483")
}
```

Hyperbolic Functions

The library also includes hyperbolic functions and their inverses:

```

test "hyperbolic functions" {

  inspect(@math sinh(1.0), content="1.1752011936438014")
  inspect(@math cosh(1.0), content="1.5430806348152437")
  inspect(@math tanh(1.0), content="0.7615941559557649")


  inspect(@math asinh(1.0), content="0.881373587019543")
  inspect(@math acosh(2.0), content="1.3169578969248166")
  inspect(@math atanh(0.5), content="0.5493061443340548")
}
```

Special Functions

Two-argument Functions

Some special mathematical functions taking two arguments:

```

test "special functions" {

  inspect(@math atan2(1.0, 1.0), content="0.7853981633974483")


  inspect(@math hypot(3.0, 4.0), content="5")


  inspect(@math cbrt(8.0), content="2")
}
```

ref

This package provides functionality for working with mutable references, allowing you to create sharable mutable values that can be modified safely.

Creating and Accessing References

References can be created using @ref.new(). The reference value can be accessed through the val field:

```

test "creating and accessing refs" {
  let r1 = @ref new(42)
  inspect(r1 val, content="42")
}
```

Updating Reference Values

The update function allows modifying the contained value using a transformation function:

```

test "updating refs" {
  let counter = @ref new(0)
  counter update(x => x + 1)
  inspect(counter val, content="1")
  counter update(x => x * 2)
  inspect(counter val, content="2")
}
```

Mapping References

The map function transforms a reference while preserving the reference wrapper:

```

test "mapping refs" {
  let num = @ref new(10)
  let doubled = num map(x => x * 2)
  inspect(doubled val, content="20")
  let squared = num map(x => x * x)
  inspect(squared val, content="100")
}
```

Swapping Reference Values

You can exchange the values of two references using the swap function:

```

test "swapping refs" {
  let r1 = @ref new("first")
  let r2 = @ref new("second")
  @ref swap(r1, r2)
  inspect(r1 val, content="second")
  inspect(r2 val, content="first")
}
```

Temporary Value Protection

The protect function temporarily sets a reference to a value and restores it aft
er executing a block:

```

test "protected updates" {
  let state = @ref new(100)
  let mut middle = 0
  let result = state protect(50, () => {
    middle = state val
    42
  })
  inspect(middle, content="50")
  inspect(result, content="42")
  inspect(state val, content="100")
}
```

This is useful for temporarily modifying state that needs to be restored afterwa
rds.

bytes

This package provides utilities for working with sequences of bytes, offering bo
th mutable (Bytes) and immutable (View) representations.

Creating Bytes

You can create Bytes from various sources including arrays, fixed arrays, and it
erators:

```

test "bytes creation" {

  let arr = [b'h', b'e', b'l', b'l', b'o']
  let bytes1 = @bytes.from_array(arr)
  inspect(
    bytes1,
    content=(
      #|b"\x68\x65\x6c\x6c\x6f"
    ),
  )


  let fixed = FixedArray::make(3, b'a')
  let bytes2 = @bytes.of(fixed)
  inspect(
    bytes2,
    content=(
      #|b"\x61\x61\x61"
    ),
  )


  let empty = @bytes.default()
  inspect(
    empty,
    content=(
      #|b""
    ),
  )


  let iter_bytes = @bytes.from_iter(arr.iter())
  inspect(
    iter_bytes,
    content=(
      #|b"\x68\x65\x6c\x6c\x6f"
    ),
  )
}
```

Converting Between Formats

Bytes can be converted to and from different formats:

```
test "bytes conversion" {
  let original = [b'x', b'y', b'z']
  let bytes = @bytes from_array(original)


  let array = bytes to_array()
  inspect(array, content="[b'\\x78', b'\\x79', b'\\x7A']")


  let fixed = bytes to_fixedarray()
  inspect(fixed, content="[b'\\x78', b'\\x79', b'\\x7A']")


  let collected = bytes iter() to_array()
  inspect(collected, content="[b'\\x78', b'\\x79', b'\\x7A']")
}
```

Working with Views

Views provide a way to work with portions of bytes and interpret them as various numeric types:

```
test "bytes view operations" {

  let num_bytes = @bytes from_array([0x12, 0x34, 0x56, 0x78])


  let view = num_bytes[:]


  inspect(view[0], content="b'\\x12'")


  inspect(view to_int_be(), content="305419896")


  inspect(view to_int_le(), content="2018915346")


  let sub_view = view[1:3]
  inspect(sub_view length(), content="2")
}
```

Binary Data Interpretation

Views provide methods to interpret byte sequences as various numeric types in bo
th little-endian and big-endian formats:

```

test "numeric interpretation" {

  let int64_bytes = @bytes from_array([
    0x00, 0x00, 0x00, 0x00, 0x00, 0x00, 0x00, 0x42,
  ])
  let int64_view = int64_bytes[:]
  inspect(int64_view to_int64_be(), content="66")
  inspect(int64_view to_uint64_le(), content="4755801206503243776")
}
```

Concatenation and Comparison

Bytes can be concatenated and compared:

```

test "bytes operations" {
  let b1 = @bytes from_array([b'a', b'b'])
  let b2 = @bytes from_array([b'c', b'd'])


  let combined = b1 + b2
  inspect(
    combined,
    content=(
      #|b"\x61\x62\x63\x64"
    ),
  )


  let same = @bytes from_array([b'a', b'b'])
  let different = @bytes from_array([b'x', b'y'])
  inspect(b1 == same, content="true")
  inspect(b1 == different, content="false")
  inspect(b1 < b2, content="true")
}
```

char

This package provides a set of utilities for working with characters, focusing o
n character classification and validation.

Basic ASCII Classification

Functions for determining if a character belongs to various ASCII categories.

```

test "ascii classification" {

  inspect('A' is_ascii(), content="true")
  inspect('?' is_ascii(), content="false")


  inspect('Z' is_ascii_alphabetic(), content="true")
  inspect('1' is_ascii_alphabetic(), content="false")


  inspect('A' is_ascii_uppercase(), content="true")
  inspect('a' is_ascii_uppercase(), content="false")
  inspect('a' is_ascii_lowercase(), content="true")
  inspect('A' is_ascii_lowercase(), content="false")
}
```

Number Classification

Functions for identifying digits in different number bases.

```

test "number classification" {

  inspect('5' is_ascii_digit(), content="true")
  inspect('x' is_ascii_digit(), content="false")


  inspect('F' is_ascii_hexdigit(), content="true")
  inspect('G' is_ascii_hexdigit(), content="false")


  inspect('7' is_ascii_octdigit(), content="true")
  inspect('8' is_ascii_octdigit(), content="false")


  inspect('5' is_digit(6U), content="true")
  inspect('6' is_digit(6U), content="false")


  inspect('1' is_numeric(), content="true")
  inspect('A' is_numeric(), content="false")
}
```

Special Characters

Functions for identifying whitespace, control characters and other special chara
cters.

```

test "special characters" {

  inspect(’ ’ is_ascii_whitespace(), content="true")
  inspect(’\n’ is_whitespace(), content="true")


  inspect(’\u0000’ is_ascii_control(), content="true")
  inspect(’\u007F’ is_control(), content="true")


  inspect(’!’ is_ascii_graphic(), content="true")
  inspect(’ ’ is_ascii_graphic(), content="false")
  inspect(’,’ is_ascii_punctuation(), content="true")
}
```

Method Style Usage

All character classification functions can also be called as methods directly on characters.

```

test "method style" {

  let c = ’A’
  inspect(c is_ascii(), content="true")
  inspect(c is_ascii_alphabetic(), content="true")
  inspect(c is_ascii_uppercase(), content="true")


  let d = ’7’
  inspect(d is_ascii_digit(), content="true")
  inspect(d is_digit(8U), content="true")
  inspect(d is_ascii_hexdigit(), content="true")


  let s = ’ ’
  inspect(s is_ascii_whitespace(), content="true")
  inspect(s is_whitespace(), content="true")
}
```

Deque

Deque is a double-ended queue implemented as a round-robin queue, supporting O(1) head or tail insertion and querying, just like double-ended queues in other languages(C++ std::deque / Rust VecDeque), here deque also supports random access.

Usage

Create

You can create a deque manually via the new() or construct it using the of()
.

```

test {
  let _dv : @deque Deque[Int] = @deque new()
  let _dv = @deque of([1, 2, 3, 4, 5])

}
```

If you want to set the length at creation time to minimize expansion consumption
, you can add parameter capacity to the new() function.

```

test {
  let _dv : @deque Deque[Int] = @deque new(capacity=10)

}
```

Length & Capacity

A deque is an indefinite-length, auto-expandable datatype. You can use length() Tj T* () to get the number of eler
 current capacity.

```

test {
  let dv = @deque of([1, 2, 3, 4, 5])
  assert_eq(dv length(), 5)
  assert_eq(dv capacity(), 5)
}
```

Similarly, you can use the is_empty to determine whether the queue is empty.

```

test {
  let dv : @deque Deque[Int] = @deque new()
  assert_eq(dv is_empty(), true)
}
```

You can use reserve_capacity to reserve capacity, ensures that it can hold at le
ast the number of elements specified by the capacity argument.

```

test {
  let dv = @deque of([1])
  dv reserve_capacity(10)
  assert_eq(dv capacity(), 10)
}
```

Also, you can use shrink_to_fit to shrink the capacity of the deque.

```

test {
  let dv = @deque new(capacity=10)
  dv push_back(1)
  dv push_back(2)
  dv push_back(3)
  assert_eq(dv capacity(), 10)
  dv shrink_to_fit()
  assert_eq(dv capacity(), 3)
}
```

Front & Back & Get

You can use front() and back() to get the head and tail elements of the queu
e, respectively. Since the queue may be empty, their return values are both Opti
on, or None if the queue is empty.

```

test {
  let dv = @deque of([1, 2, 3, 4, 5])
  assert_eq(dv front(), Some(1))
  assert_eq(dv back(), Some(5))
}
```

You can also use get to access elements of the queue directly, but be careful no
t to cross the boundaries!

```

test {
  let dv = @deque of([1, 2, 3, 4, 5])
  assert_eq(dv[0], 1)
  assert_eq(dv[4], 5)
}
```

Push & Set

Since the queue is bi-directional, you can use push_front() and push_back()
to add values to the head or tail of the queue, respectively.

```

test {
  let dv = @deque of([1, 2, 3, 4, 5])
  dv push_front(6)
  dv push_front(7)
  dv push_back(8)
  dv push_back(9)

}
```

You can also use Deque::set or operator _[_]=_to set elements of the queue direc
tly, but be careful not to cross the boundaries!

```

test {
  let dv = @deque of([1, 2, 3, 4, 5])
  dv[0] = 5
  assert_eq(dv[0], 5)
}
```

Pop

You can use pop_front() and pop_back() to pop the element at the head or tai
l of the queue, respectively, and like [Front & Back](#Front & Back & Get), th
eir return values are Option, loaded with the value of the element being popped.

```

test {
  let dv = @deque of([1, 2, 3, 4, 5])
  let _back = dv pop_back()
  assert_eq(dv back(), Some(4))
  let _front = dv pop_front()
  assert_eq(dv front(), Some(2))
  assert_eq(dv length(), 3)
}
```

If you only want to pop an element without getting the return value, you can use
 unsafe_pop_front() with unsafe_pop_back(). These two functions will panic i
f the queue is empty.

```

test {
  let dv = @deque of([1, 2, 3, 4, 5])
  dv unsafe_pop_front()
  assert_eq(dv front(), Some(2))
  dv unsafe_pop_back()
  assert_eq(dv back(), Some(4))
}
```

Clear

You can use clear to clear a deque. But note that the memory it already occupies
 does not change.

```

test {
  let dv = @deque of([1, 2, 3, 4, 5])
  dv clear()
  assert_eq(dv is_empty(), true)
}
```

Equal

deque supports comparing them directly using equal.

```

test {
  let dqa = @deque of([1, 2, 3, 4, 5])
  let dqb = @deque of([1, 2, 3, 4, 5])
  assert_eq(dqa, dqb)
}
```

Iter & Map

deque supports vector-like iter/iteri/map/mapi functions and their inverse forms
.

```

test {
  let dv = @deque of([1, 2, 3, 4, 5])
  let arr = []
  dv each(elem => arr push(elem))
  assert_eq(arr, [1, 2, 3, 4, 5])
  let arr2 = []
  dv eachi((i, _elem) => arr2 push(i))
  assert_eq(arr2, [0, 1, 2, 3, 4])
  let arr3 = []
  let _ = dv map(elem => arr3 push(elem + 1))
  assert_eq(arr3, [2, 3, 4, 5, 6])
  let arr4 = []
  let _ = dv mapi((i, elem) => arr4 push(elem + i))
  assert_eq(arr4, [1, 3, 5, 7, 9])
}
```

Search & Contains

You can use contains() to find out if a value is in the deque, or search() t
o find its index in the deque.

```

test {
  let dv = @deque of([1, 2, 3, 4, 5])
  assert_eq(dv contains(1), true)
  assert_eq(dv contains(6), false)
  assert_eq(dv search(1), Some(0))
  assert_eq(dv search(6), None)
}
```

Error Package Documentation

This package provides utilities for working with MoonBit’s error handling system
, including implementations of Show and ToJson traits for the built-in Error typ
e.

Basic Error Usage

MoonBit uses a structured error system with raise and try constructs:

```

test "basic error handling" {
  fn divide(a : Int, b : Int) -> Int raise {
    if b == 0 {
      raise Failure("Division by zero")
    } else {
      a / b
    }
  }


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


  let result2 = try? divide(10, 0)
  inspect(result2, content="Err(Failure(\"Division by zero\"))")
}
```

Custom Error Types

Define custom error types using suberror:

```
suberror ValidationError String


suberror NetworkError String


test "custom errors" {
  fn validate_email(email : String) -> String raise ValidationError {
    if email length() > 5 {
      email
    } else {
      raise ValidationError("Invalid email format")
    }
  }

  fn fetch_data(url : String) -> String raise NetworkError {
    if url length() > 10 {
      "data"
    } else {
      raise NetworkError("Invalid URL")
    }
  }


  let email_result = try? validate_email("short")
  match email_result {
    Ok(_) => inspect(false, content="true")
    Err(_) => inspect(true, content="true")
  }


  let data_result = try? fetch_data("short")
  match data_result {
    Ok(_) => inspect(false, content="true")
    Err(_) => inspect(true, content="true")
  }
}
```

Error Display and JSON Conversion

The error package provides Show and ToJson implementations:

```

suberror MyError Int derive(ToJson)


test "error display and json" {
  let error : Error = MyError(42)


  let error_string = error to_string()
  inspect(error_string length() > 0, content="true")


  let error_json = error to_json()
  inspect(error_json, content="Array([String(\"MyError\"), Number(42)])"
}
```

Error Propagation and Handling

Handle errors at different levels of your application:

```

suberror ParseError String


suberror FileError String


test "error propagation" {
  fn parse_number(s : String) -> Int raise ParseError {
    if s == "42" {
      42
    } else {
      raise ParseError("Invalid number: " + s)
    }
  }

  fn read_and_parse(content : String) -> Int raise {
    parse_number(content) catch {
      ParseError(msg) => raise FileError("Parse failed: " + msg)
    }
  }


  let result1 = try! read_and_parse("42")
  inspect(result1, content="42")


  let result2 = try? read_and_parse("invalid")
  match result2 {
    Ok(_) => inspect(false, content="true")
    Err(_) => inspect(true, content="true")
  }
}
```

Resource Management with Finally

Use protect functions for resource cleanup:

```

suberror ResourceError String


test "resource management" {
  fn risky_operation() -> String raise ResourceError {
    raise ResourceError("Something went wrong")
  }


  fn use_resource() -> String raise {

    risky_operation() catch {
      ResourceError(_) =>

        raise Failure("Operation failed after cleanup")
    }
  }

  let result = try? use_resource()
  match result {
    Ok(_) => inspect(false, content="true")
    Err(_) => inspect(true, content="true")
  }
}
```

Error Composition

Combine multiple error-producing operations:

```

suberror ConfigError String


suberror DatabaseError String


test "error composition" {
  fn load_config() -> String raise ConfigError {
    if true {
      "config_data"
    } else {
      raise ConfigError("Config not found")
    }
  }

  fn connect_database(config : String) -> String raise DatabaseError {
    if config == "config_data" {
      "connected"
    } else {
      raise DatabaseError("Invalid config")
    }
  }

  fn initialize_app() -> String raise {
    let config = load_config() catch {
      ConfigError(msg) => raise Failure("Config error: " + msg)
    }
    let db = connect_database(config) catch {
      DatabaseError(msg) => raise Failure("Database error: " + msg)
    }
    "App initialized with " + db
  }

  let app_result = try! initialize_app()
  inspect(app_result, content="App initialized with connected")
}
```

Best Practices

- **Use specific error types**: Create custom suberror types for different error categ
ories
- **Provide meaningful messages**: Include context and actionable information in error
 messages
- **Handle errors at appropriate levels**: Don’t catch errors too early; let them prop
agate to where they can be properly handled
- **Use** try! for operations that should not fail: This will panic if an error occurs
, making failures visible during development
- **Use** try? for recoverable errors: This returns a Result type that can be pattern
matched
- **Implement proper cleanup**: Use the protect pattern or similar constructs for reso
urce management

Performance Notes

- Error handling in MoonBit is zero-cost when no errors occur
- Error propagation is efficient and doesn’t require heap allocation for the error path
- Custom error types with derive(ToJson) automatically generate efficient JSON s
erialization

HashSet

A mutable hash set based on a Robin Hood hash table.

Usage

Create

You can create an empty set using new() or construct it using from_array().

```

test {
  let _set1 = @hashset of([1, 2, 3, 4, 5])
  let _set2 : @hashset HashSet[String] = @hashset new()

}
```

Insert & Contain

You can use insert() to add a key to the set, and contains() to check whethe
r a key exists.

```

test {
  let set : @hashset HashSet[String] = @hashset new()
  set add("a")
  assert_eq(set contains("a"), true)
}
```

Remove

You can use remove() to remove a key.

```

test {
  let set = @hashset of(["a", "b", "c"])
  set remove("a")
  assert_eq(set contains("a"), false)
}
```

Size & Capacity

You can use size() to get the number of keys in the set, or capacity() to get the current capacity.

```

test {
  let set = @hashset of(["a", "b", "c"])
  assert_eq(set size(), 3)
  assert_eq(set capacity(), 8)
}
```

Similarly, you can use is_empty() to check whether the set is empty.

```

test {
  let set : @hashset HashSet[Int] = @hashset new()
  assert_eq(set is_empty(), true)
}
```

Clear

You can use clear to remove all keys from the set, but the allocated memory will not change.

```

test {
  let set = @hashset of(["a", "b", "c"])
  set clear()
  assert_eq(set is_empty(), true)
}
```

Iteration

You can use each() or eachi() to iterate through all keys.

```

test {
  let set = @hashset of(["a", "b", "c"])
  let arr = []
  set each(k => arr push(k))
  let arr2 = []
  set eachi((i, k) => arr2 push((i, k)))
}
```

Set Operations

You can use union(), intersection(), difference() and symmetric_difference() to perform set operations.

```

test {
  let m1 = @hashset of(["a", "b", "c"])
  let m2 = @hashset of(["b", "c", "d"])
  fn to_sorted_array(set : @hashset HashSet[String]) {
    let arr = set to_array()
    arr sort()
    arr
  }

  assert_eq(m1 union(m2) |> to_sorted_array, ["a", "b", "c", "d"])
  assert_eq(m1 intersection(m2) |> to_sorted_array, ["b", "c"])
  assert_eq(m1 difference(m2) |> to_sorted_array, ["a"])
  assert_eq(m1 symmetric_difference(m2) |> to_sorted_array, ["a", "d"])
}
```

Rational (DEPRECATED)

& þ**This module is deprecated. Use** @rational in module moonbitlang/
x instead. Note that you need to rename Rational to Rational64.

The Rational type represents a rational number, which is a number that can be ex
pressed as a fraction a/b where a and b are integers and b is not zero.

All tests and examples have been removed. Please refer to the new moonbitlang/x
module for updated documentation and examples.

bool

This package provides utility functions for working with boolean values in MoonB
it, primarily focused on type conversions that are useful in systems programming
, bitwise operations, and numerical computations.

Overview

Boolean values in MoonBit can be seamlessly converted to numeric types, followin
g the standard convention where true maps to 1 and false maps to 0. This is part
icularly useful for:

- Conditional arithmetic and accumulation
- Interfacing with C libraries or low-level code
- Implementing boolean algebra with numeric operations
- Converting logical results to flags or indices

Basic Integer Conversion

Convert boolean values to standard integers for arithmetic operations:

```
test "bool to integer conversions" {

  inspect(true to_int(), content="1")
  inspect(false to_int(), content="0")


  let score = 100
  let bonus_applied = true
  let final_score = score + bonus_applied to_int() * 50
  inspect(final_score, content="150")


  let conditions = [true, false, true, true, false]
  let count = conditions fold(init=0, fn(acc, cond) { acc + cond to_int(
  inspect(count, content="3")
}
```

Specialized Integer Types

For specific use cases requiring different integer widths and signedness:

```
test "bool to specialized integer types" {
  let flag = true
  let no_flag = false


  inspect(flag to_uint(), content="1")
  inspect(no_flag to_uint(), content="0")


  inspect(flag to_int64(), content="1")
  inspect(no_flag to_int64(), content="0")


  inspect(flag to_uint64(), content="1")
  inspect(no_flag to_uint64(), content="0")
}
```

Practical Use Cases

Boolean Indexing and Selection

```

test "boolean indexing" {

  let options = ["default", "enhanced"]
  let use_enhanced = true
  let selected = options[use_enhanced to_int()]
  inspect(selected, content="enhanced")


  let base_value = 10
  let multiplier = 2
  let apply_multiplier = false
  let result = base_value * (1 + apply_multiplier to_int() * (multiplier
  inspect(result, content="10")
}
```

Bit Manipulation and Flags

```

test "flags and bit operations" {

  let read_permission = true
  let write_permission = false
  let execute_permission = true
  let permissions = (read_permission to_uint() << 2) |
    (write_permission to_uint() << 1) |
    execute_permission to_uint()
  inspect(permissions, content="5")
}
```

Statistical and Mathematical Operations

```

test "statistical operations" {

  let test_results = [true, true, false, true, false, true, true]
  let successes = test_results fold(init=0, fn(acc, result) {
    acc + result to_int()
  })
  let total = test_results length()
  let success_rate = successes to_double() / total to_double()
  inspect(success_rate > 0.7, content="true")


  let feature_enabled = [true, false, true]
  let weights = [0.6, 0.3, 0.1]


  let score1 = feature_enabled[0] to_int() to_double() * weights[0]
  let score2 = feature_enabled[1] to_int() to_double() * weights[1]
  let score3 = feature_enabled[2] to_int() to_double() * weights[2]
  let weighted_score = score1 + score2 + score3
  inspect(weighted_score == 0.7, content="true")
}
```

This package provides the essential bridge between MoonBit's boolean logic and numeric computations, enabling elegant solutions for conditional arithmetic, flag operations, and data processing workflows.

Coverage Package Documentation

This package provides code coverage tracking utilities for MoonBit programs. It includes tools for measuring which parts of your code are executed during testing and generating coverage reports.

Coverage Counter

The core component for tracking code execution:

```

test "coverage counter basics" {

  let counter = CoverageCounter::new(5)


  inspect(counter to_string(), content="[0, 0, 0, 0, 0]")


  counter incr(0)
  counter incr(2)
  counter incr(0)


  inspect(counter to_string(), content="[2, 0, 1, 0, 0]")
}
```

Tracking Code Execution

Use coverage counters to track which code paths are executed:

```

test "tracking execution paths" {
  let counter = CoverageCounter::new(3)
  fn conditional_function(x : Int, coverage : CoverageCounter) -> String
    if x > 0 {
      coverage incr(0)
      "positive"
    } else if x < 0 {
      coverage incr(1)
      "negative"
    } else {
      coverage incr(2)
      "zero"
    }
  }


  let result1 = conditional_function(5, counter)
  inspect(result1, content="positive")
  let result2 = conditional_function(-3, counter)
  inspect(result2, content="negative")
  let result3 = conditional_function(0, counter)
  inspect(result3, content="zero")


  inspect(counter to_string(), content="[1, 1, 1]")
}
```

Loop Coverage Tracking

Track coverage in loops and iterations:

```

test "loop coverage" {
  let counter = CoverageCounter::new(2)
  fn process_array(arr : Array[Int], coverage : CoverageCounter) -> Int
    let mut sum = 0
    for x in arr {
      if x % 2 == 0 {
        coverage incr(0)
        sum = sum + x
      } else {
        coverage incr(1)
        sum = sum + x * 2
      }
    }
    sum
  }

  let test_data = [1, 2, 3, 4, 5]
  let result = process_array(test_data, counter)


  inspect(result, content="24")


  let coverage_str = counter to_string()
  inspect(coverage_str length() > 5, content="true")
}
```

Function Coverage

Track coverage across different functions:

```

test "function coverage" {
  let counter = CoverageCounter::new(4)
  fn math_operations(
    a : Int,
    b : Int,
    op : String,
    coverage : CoverageCounter,
  ) -> Int {
    match op {
      "add" => {
        coverage incr(0)
        a + b
      }
      "sub" => {
        coverage incr(1)
        a - b
      }
      "mul" => {
        coverage incr(2)
        a * b
      }
      _ => {
        coverage incr(3)
        0
      }
    }
  }


  let add_result = math_operations(10, 5, "add", counter)
  inspect(add_result, content="15")
  let sub_result = math_operations(10, 5, "sub", counter)
  inspect(sub_result, content="5")
  let unknown_result = math_operations(10, 5, "unknown", counter)
  inspect(unknown_result, content="0")


  let final_coverage = counter to_string()
  inspect(final_coverage, content="[1, 1, 0, 1]")
}
```

Coverage Analysis

Analyze coverage data to understand code execution:

```

test "coverage analysis" {
  let counter = CoverageCounter::new(6)
  fn complex_function(input : Int, coverage : CoverageCounter) -> String
    coverage incr(0)
    if input < 0 {
      coverage incr(1)
      return "negative"
    }
    coverage incr(2)
    if input == 0 {
      coverage incr(3)
      return "zero"
    }
    coverage incr(4)
    if input > 100 {
      coverage incr(5)
      "large"
    } else {
      "small"
    }
  }


  let result1 = complex_function(-5, counter)
  inspect(result1, content="negative")
  let result2 = complex_function(0, counter)
  inspect(result2, content="zero")
  let result3 = complex_function(50, counter)
  inspect(result3, content="small")


  let coverage = counter to_string()

  inspect(coverage length() > 10, content="true")
}
```

Integration with Testing

Coverage tracking integrates with MoonBit’s testing system:

```

test "testing integration" {



  fn test_function_with_coverage() -> Bool {

    let counter = CoverageCounter::new(2)
    fn helper(condition : Bool, cov : CoverageCounter) -> String {
      if condition {
        cov incr(0)
        "true_branch"
      } else {
        cov incr(1)
        "false_branch"
      }
    }


    let result1 = helper(true, counter)
    let result2 = helper(false, counter)
    result1 == "true_branch" && result2 == "false_branch"
  }

  let test_passed = test_function_with_coverage()
  inspect(test_passed, content="true")
}
```

Coverage Reporting

Generate and analyze coverage reports:

```

test "coverage reporting" {
  let counter = CoverageCounter::new(3)


  counter incr(0)
  counter incr(0)
  counter incr(2)


  let report = counter to_string()
  inspect(report, content="[2, 0, 1]")


  fn analyze_coverage(_coverage_str : String) -> (Int, Int) {


    (2, 3)
  }

  let (covered, total) = analyze_coverage(report)
  inspect(covered, content="2")
  inspect(total, content="3")
}
```

Best Practices

1. Automatic Coverage Generation

In real applications, coverage tracking is typically generated automatically:

```


fn example_function(x : Int) -> String {

  if x > 0 {

    "positive"
  } else {

    "non-positive"
  }

}


test "automatic coverage concept" {
  let result = example_function(5)
  inspect(result, content="positive")
}
```

2. Coverage-Driven Testing

Use coverage information to improve test quality:

```

test "coverage driven testing" {

  fn multi_branch_function(a : Int, b : Int) -> String {
    if a > b {
      "greater"
    } else if a < b {
      "less"
    } else {
      "equal"
    }
  }


  inspect(multi_branch_function(5, 3), content="greater")
  inspect(multi_branch_function(2, 7), content="less")
  inspect(multi_branch_function(4, 4), content="equal")


}
```

Integration with Build System

Coverage tracking integrates with MoonBit's build tools:

- Use moon test to run tests with coverage tracking
- Use moon coverage analyze to generate coverage reports
- Coverage data helps identify untested code paths
- Supports both line coverage and branch coverage analysis

Performance Considerations

- Coverage tracking adds minimal runtime overhead
- Counters use efficient fixed arrays for storage
- Coverage instrumentation is typically removed in release builds
- Use coverage data to optimize test suite performance

Common Use Cases

- **Test Quality Assessment**: Ensure comprehensive test coverage
- **Dead Code Detection**: Find unused code paths
- **Regression Testing**: Verify that tests exercise the same code paths
- **Performance Analysis**: Identify frequently executed code for optimization
- **Code Review**: Understand which parts of code are well-tested

The coverage package provides essential tools for maintaining high-quality, well-tested MoonBit code through comprehensive coverage analysis.

MoonBit Float Package Documentation

This package provides operations on 32-bit floating-point numbers (Float). It includes basic arithmetic, trigonometric functions, exponential and logarithmic functions, as well as utility functions for rounding and conversion.

Special Values

The package defines several special floating-point values:

```

test "special float values" {

  inspect(@float infinity, content="Infinity")
  inspect(@float neg_infinity, content="-Infinity")


  inspect(@float not_a_number, content="NaN")


  inspect(@float max_value, content="3.4028234663852886e+38")
  inspect(@float min_value, content="-3.4028234663852886e+38")
  inspect(@float min_positive, content="1.1754943508222875e-38")
}


test "checking special values" {

  inspect(@float infinity is_inf(), content="true")
  inspect(@float neg_infinity is_neg_inf(), content="true")
  inspect(@float infinity is_pos_inf(), content="true")
  inspect(@float not_a_number is_nan(), content="true")
}
```

Rounding Functions

The package provides various ways to round floating-point numbers:

```

test "rounding functions" {

  inspect(@float ceil(3.2), content="4")
  inspect(@float ceil(-3.2), content="-3")


  inspect(@float floor(3.2), content="3")
  inspect(@float floor(-3.2), content="-4")


  inspect(@float round(3.7), content="4")
  inspect(@float round(3.2), content="3")


  inspect(@float trunc(3.7), content="3")
  inspect(@float trunc(-3.7), content="-3")
}
```

Utility Functions

Other useful operations on floats:

```

test "utility functions" {

  inspect(@float abs(-3.14), content="3.140000104904175")


  inspect(3.14 to_int(), content="3")


  inspect(@float default(), content="0")
}
```

Byte Representation

Functions to convert floats to their byte representation:

```

test "byte representation" {
  let x : Float = 3.14

  let be_bytes = x to_be_bytes()

  let le_bytes = x to_le_bytes()
  inspect(be_bytes length(), content="4")
  inspect(le_bytes length(), content="4")
}
```

Method Style

All functions can also be called in method style:

```

test "method style calls" {
  let x : Float = 3.14
  inspect(x floor(), content="3")
  inspect(x ceil(), content="4")
  inspect(x round(), content="3")
  let y : Float = 2.0
  inspect(y pow(3.0), content="8")
}
```

int16

This package provides a fixed-width 16-bit signed integer type.

Range and Constants

The Int16 type represents values from -32768 to 32767 (inclusive). The package provides these boundary values as constants:

```

test "int16 range" {
  inspect(@int16 min_value, content="-32768")
  inspect(@int16 max_value, content="32767")
}
```

Arithmetic Operations

The Int16 type supports standard arithmetic operations:

```

test "int16 arithmetic" {
  let a : Int16 = 100
  let b : Int16 = 50


  inspect(a + b, content="150")
  inspect(a - b, content="50")
  inspect(a * b, content="5000")
  inspect(a / b, content="2")


  let max = @int16 max_value
  let min = @int16 min_value
  inspect(max + 1, content="-32768")
  inspect(min - 1, content="32767")
}
```

Bitwise Operations

Int16 supports standard bitwise operations:

```

test "int16 bitwise" {
  let a : Int16 = 0b1100
  let b : Int16 = 0b1010


  inspect(a & b, content="8")
  inspect(a | b, content="14")
  inspect(a ^ b, content="6")


  let x : Int16 = 8
  inspect(x << 1, content="16")
  inspect(x >> 1, content="4")
}
```

Comparison Operations

Int16 implements the Compare trait for total ordering:

```

test "int16 comparison" {
  let a : Int16 = 100
  let b : Int16 = 50
  let c : Int16 = 100


  inspect(a == b, content="false")
  inspect(a == c, content="true")


  inspect(a > b, content="true")
  inspect(b < c, content="true")


  inspect(a compare(b), content="1")
  inspect(b compare(c), content="-1")
  inspect(a compare(c), content="0")
}
```

Default Value

Int16 implements the Default trait, with 0 as its default value:

```

test "int16 default" {
  let x = Int16::default()
  inspect(x, content="0")
}
```

Type Coercion and Conversion

Integer literals can be coerced to Int16 when the type is explicitly specified:

```

test "int16 coercion" {
  let a : Int16 = 42
  let b : Int16 = 0xFF
  let c : Int16 = 0b1111
  inspect(a, content="42")
  inspect(b, content="255")
  inspect(c, content="15")
}
```

Set Package Documentation

This package provides a hash-based set data structure that maintains insertion o
rder. The Set[K] type stores unique elements and provides efficient membership t
esting, insertion, and deletion operations.

Creating Sets

There are several ways to create sets:

```

test "creating sets" {

  let empty_set : @set Set[Int] = @set Set::new()
  inspect(empty_set size(), content="0")
  inspect(empty_set is_empty(), content="true")


  let set_with_capacity : @set Set[Int] = @set Set::new(capacity=16)
  inspect(set_with_capacity capacity(), content="16")


  let from_array = @set Set::from_array([1, 2, 3, 2, 1])
  inspect(from_array size(), content="3")


  let from_fixed = @set Set::of([10, 20, 30])
  inspect(from_fixed size(), content="3")


  let from_iter = @set Set::from_iter([1, 2, 3, 4, 5] iter())
  inspect(from_iter size(), content="5")
}
```

Basic Operations

Add, remove, and check membership:

```
test "basic operations" {
  let set = @set Set::new()


  set add("apple")
  set add("banana")
  set add("cherry")
  inspect(set size(), content="3")


  set add("apple")
  inspect(set size(), content="3")


  inspect(set contains("apple"), content="true")
  inspect(set contains("orange"), content="false")


  set remove("banana")
  inspect(set contains("banana"), content="false")
  inspect(set size(), content="2")


  let was_added = set add_and_check("date")
  inspect(was_added, content="true")
  let was_added_again = set add_and_check("date")
  inspect(was_added_again, content="false")
  let was_removed = set remove_and_check("cherry")
  inspect(was_removed, content="true")
  let was_removed_again = set remove_and_check("cherry")
  inspect(was_removed_again, content="false")
}
```

Set Operations

Perform mathematical set operations:

```
test "set operations" {
  let set1 = @set Set::from_array([1, 2, 3, 4])
  let set2 = @set Set::from_array([3, 4, 5, 6])


  let union_set = set1 union(set2)
  let union_array = union_set to_array()
  inspect(union_array length(), content="6")


  let union_alt = set1 | set2
  inspect(union_alt size(), content="6")


  let intersection_set = set1 intersection(set2)
  let intersection_array = intersection_set to_array()
  inspect(intersection_array length(), content="2")


  let intersection_alt = set1 & set2
  inspect(intersection_alt size(), content="2")


  let difference_set = set1 difference(set2)
  let difference_array = difference_set to_array()
  inspect(difference_array length(), content="2")


  let difference_alt = set1 - set2
  inspect(difference_alt size(), content="2")


  let sym_diff_set = set1 symmetric_difference(set2)
  let sym_diff_array = sym_diff_set to_array()
  inspect(sym_diff_array length(), content="4")


  let sym_diff_alt = set1 ^ set2
  inspect(sym_diff_alt size(), content="4")
}
```

Set Relationships

Test relationships between sets:

```

test "set relationships" {
  let small_set = @set Set::from_array([1, 2])
  let large_set = @set Set::from_array([1, 2, 3, 4])
  let disjoint_set = @set Set::from_array([5, 6, 7])


  inspect(small_set is_subset(large_set), content="true")
  inspect(large_set is_subset(small_set), content="false")


  inspect(large_set is_superset(small_set), content="true")
  inspect(small_set is_superset(large_set), content="false")


  inspect(small_set is_disjoint(disjoint_set), content="true")
  inspect(small_set is_disjoint(large_set), content="false")


  let set1 = @set Set::from_array([1, 2, 3])
  let set2 = @set Set::from_array([3, 2, 1])
  inspect(set1 == set2, content="true")
}
```

Iteration and Conversion

Iterate over sets and convert to other types:

```

test "iteration and conversion" {
  let set = @set Set::from_array(["first", "second", "third"])


  let array = set to_array()
  inspect(array length(), content="3")


  let mut count = 0
  set each(fn(_element) { count = count + 1 })
  inspect(count, content="3")


  let mut indices_sum = 0
  set eachi(fn(i, _element) { indices_sum = indices_sum + i })
  inspect(indices_sum, content="3")


  let elements = set iter() collect()
  inspect(elements length(), content="3")


  let copied_set = set copy()
  inspect(copied_set size(), content="3")
  inspect(copied_set == set, content="true")
}
```

Modifying Sets

Clear and modify existing sets:

```

test "modifying sets" {
  let set = @set Set::from_array([10, 20, 30, 40, 50])
  inspect(set size(), content="5")


  set clear()
  inspect(set size(), content="0")
  inspect(set is_empty(), content="true")


  set add(100)
  set add(200)
  inspect(set size(), content="2")
  inspect(set contains(100), content="true")
}
```

JSON Serialization

Sets can be serialized to JSON as arrays:

```

test "json serialization" {
  let set = @set Set::from_array([1, 2, 3])
  let json = set to_json()


  inspect(json, content="Array([Number(1), Number(2), Number(3)])")


  let string_set = @set Set::from_array(["a", "b", "c"])
  let string_json = string_set to_json()
  inspect(
    string_json,
    content="Array([String(\"a\"), String(\"b\"), String(\"c\")])",
  )
}
```

Working with Different Types

Sets work with any type that implements Hash and Eq:

```

test "different types" {

  let int_set = @set Set::from_array([1, 2, 3, 4, 5])
  inspect(int_set contains(3), content="true")


  let string_set = @set Set::from_array(["hello", "world", "moonbit"])
  inspect(string_set contains("world"), content="true")



  let char_codes = @set Set::from_array([97, 98, 99])
  inspect(char_codes contains(98), content="true")


  let bool_codes = @set Set::from_array([1, 0, 1])
  inspect(bool_codes size(), content="2")
}
```

Performance Examples

Demonstrate efficient operations:

```

test "performance examples" {

  let large_set = @set Set::new(capacity=1000)


  for i in 0..<100 {
    large_set add(i)
  }
  inspect(large_set size(), content="100")


  inspect(large_set contains(50), content="true")
  inspect(large_set contains(150), content="false")


  let another_set = @set Set::new()
  for i in 50..<150 {
    another_set add(i)
  }
  let intersection = large_set intersection(another_set)
  inspect(intersection size(), content="50")
}
```

Use Cases

Sets are particularly useful for:

- **Removing duplicates**: Convert arrays to sets and back to remove duplicates
- **Membership testing**: Fast O(1) average-case lookups
- **Mathematical operations**: Union, intersection, difference operations
- **Unique collections**: Maintaining collections of unique items
- **Algorithm implementation**: Graph algorithms, caching, etc.

Performance Characteristics

- **Insertion**: O(1) average case, O(n) worst case
- **Removal**: O(1) average case, O(n) worst case
- **Lookup**: O(1) average case, O(n) worst case
- **Space complexity**: O(n) where n is the number of elements
- **Iteration order**: Maintains insertion order (linked hash set)

Best Practices

- **Pre-size when possible**: Use @set.Set::new(capacity=n) if you know the approxim
ate size
- **Use appropriate types**: Ensure your key type has good Hash and Eq implementations
- **Prefer set operations**: Use built-in union, intersection, etc. instead of manual
loops
- **Check return values**: Use add_and_check and remove_and_check when you need to kno
w if the operation succeeded
- **Consider memory usage**: Sets have overhead compared to arrays for small collectio
ns

buffer

The buffer package provides a flexible byte buffer implementation for efficient
binary data handling and serialization.

Basic Usage

Create a new buffer and write basic data:

```

test "basic buffer operations" {
  let buf = @buffer new()


  buf..write_byte(b’H’)..write_byte(b’i’)


  inspect(buf is_empty(), content="false")
  inspect(buf length(), content="2")


  let bytes = buf contents()
  inspect(
    bytes,
    content=(
      #|b"\x48\x69"
    ),
  )


  buf reset()
  inspect(buf is_empty(), content="true")
}
```

Writing Numbers

Write numbers in different encodings:

```

test "number serialization" {
  inspect(
    @buffer new()

    ..write_int_be(42)
    ..write_int_le(42)
     to_bytes(),
    content=(
      #|b"\x00\x00\x00\x2a\x2a\x00\x00\x00"
    ),
  )
  inspect(
    @buffer new()

    ..write_float_be(3.14)
    ..write_float_le(3.14)
     to_bytes(),
    content=(
      #|b"\x40\x48\xf5\xc3\xc3\xf5\x48\x40"
    ),
  )
  inspect(
    @buffer new()

    ..write_int64_be(0xAABBCCDDEEL)
    ..write_int64_le(0xAABBCCDDEEL)
     to_bytes(),
    content=(
      #|b"\x00\x00\x00\xaa\xbb\xcc\xdd\xee\xee\xdd\xcc\xbb\xaa\x00\x00\x
    ),
  )
  inspect(
    @buffer new()

    ..write_uint_be(0x2077U)
    ..write_uint_le(0x2077U)
     to_bytes(),
    content=(
      #|b"\x00\x00\x20\x77\x77\x20\x00\x00"
    ),
  )
}
```

Writing Byte Sequences

Write sequences of bytes:

```
test "byte sequence writing" {
  let buf = @buffer new()


  let bytes = b"Hello"
  buf write_bytes(bytes)


  buf write_iter(bytes iter())
  let contents = buf to_bytes()
  inspect(
    contents,
    content=(
      #|b"\x48\x65\x6c\x6c\x6f\x48\x65\x6c\x6c\x6f"
    ),
  )
}
```

Writing Structured Data

Write structured data that implements Show:

```
test "object writing" {
  let buf = @buffer new()


  buf write_object(42)


  let contents = buf contents()
  inspect(
    contents,
    content=(
      #|b"\x34\x00\x32\x00"
    ),
  )
}
```

Size Hints

Provide size hints for better performance:

```

test "buffer with size hint" {

  let buf = @buffer new(size_hint=1024)


  for i in 0..<100 {
    buf write_int_le(i)
  }


  inspect(buf length(), content="400")
}
```

Buffer as Logger

The buffer implements the Logger trait for Show:

```

test "buffer as logger" {
  let buf = @buffer new()
  let array = [1, 2, 3]


  array output(buf)
  let contents = buf contents()
  inspect(
    contents,
    content=(
      #|b"\x5b\x00\x31\x00\x2c\x00\x20\x00\x32\x00\x2c\x00\x20\x00\x33\x
    ),
  )
}
```

Converting to String/Bytes

Methods for converting buffer contents:

```

test "buffer conversion" {
  let buf = @buffer new()
  buf write_byte(b'a')
  buf write_byte(b'b')
  buf write_byte(b'c')
  let bytes = buf to_bytes()
  inspect(
    bytes,
    content=(
      #|b"\x61\x62\x63"
    ),
  )
}
```

Binary Viewing

Support for viewing subsets of bytes:

```

test "byte view writing" {
  let buf = @buffer new()
  let bytes = b"Hello World"


  buf write_bytesview(bytes[0:5])
  let contents = buf to_bytes()
  inspect(
    contents,
    content=(
      #|b"\x48\x65\x6c\x6c\x6f"
    ),
  )
}
```

Random

This is an efficient random number generation function based on the paper Fast R andom Integer Generation in an Interval by Daniel Lemire, as well as the Golang' s rand/v2 package.

Internally, it uses the Chacha8 cipher to generate random numbers. It is a crypt ographically secure pseudo-random number generator (CSPRNG) that is also very fast.

Usage

```
test {
  let r = @random Rand::new()
  assert_eq(r uint(limit=10), 7)
  assert_eq(r uint(limit=10), 0)
  assert_eq(r uint(limit=10), 5)
  assert_eq(r int(), 1064320769)
  assert_eq(r double(), 0.3318940049218405)
  assert_eq(r int(limit=10), 0)
  assert_eq(r uint(), 311122750)
  assert_eq(r int64(), 2043189202271773519)
  assert_eq(r int64(limit=10), 8)
  assert_eq(r uint64(), 3951155890335085418)
  let a = [1, 2, 3, 4, 5]
  r shuffle(a length(), (i, j) => {
    let t = a[i]
    a[i] = a[j]
    a[j] = t
  })
  assert_eq(a, [2, 1, 4, 3, 5])
}
```

Strconv

This package implements conversions to and from string representations of basic data types.

Usage

Parse

Use parse_bool, parse_double, parse_int, and parse_int64 convert strings to valu es.

```
test {
  let b = @strconv parse_bool("true")
  assert_eq(b, true)
  let i1 = @strconv parse_int("1234567")
  assert_eq(i1, 1234567)
  let i2 = @strconv parse_int("101", base=2)
  assert_eq(i2, 5)
  let d = @strconv parse_double("123.4567")
  assert_eq(d, 123.4567)
}
```

For types that implement the FromStr trait, you can also use helper function par se to convert a string to a value.

```

test {
  let a : Int = @strconv parse("123")
  assert_eq(a, 123)
  let b : Bool = @strconv parse("true")
  assert_eq(b, true)
}
```

Option

The Option type is a built-in type in MoonBit that represents an optional value.
 The type annotation Option[A] can also be written as A?.

It is an enum with two variants: Some(T), which represents a value of type T,
and None, representing no value.

Note that some methods of the Option are defined in the core/builtin package.

Usage

Create

You can create an Option value using the Some and None constructors, remember to
 give proper type annotations.

```

test {
  let some : Int? = Some(42)
  let none : String? = None
  inspect(some, content="Some(42)")
  inspect(none, content="None")
}
```

Extracting values

You can extract the value from an Option using the match expression (Pattern Ma
tching).

```

test {
  let i = Some(42)
  let j = match i {
    Some(value) => value
    None => abort("unreachable")
  }
  assert_eq(j, 42)
}
```

Or using the unwrap method, which will panic if the result is None and return th
e value if it is Some.

```

test {
  let some : Int? = Some(42)
  let value = some unwrap()
  assert_eq(value, 42)
}
```

A safer alternative to unwrap is the or method, which returns the value if it is Some, otherwise, it returns the default value.

```

test {
  let none : Int? = None
  let value = none unwrap_or(0)
  assert_eq(value, 0)
}
```

There is also the or_else method, which returns the value if it is Some, otherwise, it returns the result of the provided function.

```

test {
  let none : Int? = None
  let value = none unwrap_or_else(() => 0)
  assert_eq(value, 0)
}
```

Transforming values

You can transform the value of an Option using the map method. It applies the provided function to the value if it is Some, otherwise, it returns None.

```

test {
  let some : Int? = Some(42)
  let new_some = some map((value : Int) => value + 1)
  assert_eq(new_some, Some(43))
}
```

There is a filter method that applies a predicate to the value if it is Some, otherwise, it returns None.

```

test {
  let some : Int? = Some(42)
  let new_some = some filter((value : Int) => value > 40)
  let none = some filter((value : Int) => value > 50)
  assert_eq(new_some, Some(42))
  assert_eq(none, None)
}
```

Monadic operations

You can chain multiple operations that return Option using the bind method, which applies a function to the value if it is Some, otherwise, it returns None. Different from map, the function in argument returns an Option.

```

test {
  let some : Int? = Some(42)
  let new_some = some bind((value : Int) => Some(value + 1))
  assert_eq(new_some, Some(43))
}
```

Sometimes we want to reduce the nested Option values into a single Option, you can use the flatten method to achieve this. It transforms Some(Some(value)) into Some(value), and None otherwise.

```

test {
  let some : Int?? = Some(Some(42))
  let new_some = some flatten()
  assert_eq(new_some, Some(42))
  let none : Int?? = Some(None)
  let new_none = none flatten()
  assert_eq(new_none, None)
}
```

uint64

The moonbitlang/core/uint64 package provides functionality for working with 64-bit unsigned integers. This package includes constants, operators, and conversions for UInt64 values.

Constants

The package defines the minimum and maximum values for UInt64:

```

test "UInt64 constants" {

  inspect(@uint64 min_value, content="0")


  inspect(@uint64 max_value, content="18446744073709551615")
}
```

Arithmetic Operations

UInt64 supports standard arithmetic operations:

```

test "UInt64 arithmetic" {
  let a : UInt64 = 100UL
  let b : UInt64 = 50UL


  inspect(a + b, content="150")


  inspect(a - b, content="50")


  inspect(a * b, content="5000")


  inspect(a / b, content="2")


  inspect(@uint64 max_value + 1UL, content="0")
  inspect(@uint64 min_value - 1UL, content="18446744073709551615")
}
```

Bitwise Operations

UInt64 supports various bitwise operations:

```

test "UInt64 bitwise operations" {
  let a : UInt64 = 0b1010UL
  let b : UInt64 = 0b1100UL


  inspect(a & b, content="8")


  inspect(a | b, content="14")


  inspect(a ^ b, content="6")


  inspect(a << 1, content="20")
  inspect(a << 2, content="40")


  inspect(a >> 1, content="5")
  inspect(b >> 2, content="3")
}
```

Comparison and Equality

UInt64 supports comparison and equality operations:

```

test "UInt64 comparison and equality" {
  let a : UInt64 = 100UL
  let b : UInt64 = 50UL
  let c : UInt64 = 100UL


  inspect(a == c, content="true")
  inspect(a != b, content="true")


  inspect(a > b, content="true")
  inspect(b < a, content="true")
  inspect(a >= c, content="true")
  inspect(c <= a, content="true")
}
```

Byte Conversion

UInt64 provides methods for converting to bytes in both big-endian and little-endian formats:

```

test "UInt64 byte conversion" {

  let be_bytes = 0x123456789ABCDEF0UL to_be_bytes()
  inspect(
    be_bytes,
    content=(
      #|b"\x12\x34\x56\x78\x9a\xbc\xde\xf0"
    ),
  )


  let le_bytes = 0x123456789ABCDEF0UL to_le_bytes()
  inspect(
    le_bytes,
    content=(
      #|b"\xf0\xde\xbc\x9a\x78\x56\x34\x12"
    ),
  )
}
```

Default Value and Hashing

UInt64 implements the Default trait:

```

test "UInt64 default value" {

  let a : UInt64 = 0UL
  inspect(a, content="0")


  let value : UInt64 = 42UL
  inspect(value hash(), content="-1962516083")
}
```

Type Conversions

UInt64 works with various conversions to and from other types:

```

test "UInt64 conversions" {

  inspect((42) to_uint64(), content="42")


  let value : UInt64 = 100UL
  inspect(value to_int(), content="100")
  let as_double = value to_double()
  inspect(as_double, content="100")


  inspect((-1) to_uint64(), content="18446744073709551615")


  let from_double = 42.0 to_uint64()
  inspect(from_double, content="42")
}
```

Working with Large Numbers

UInt64 is especially useful for applications requiring large unsigned integers:

```

test "UInt64 for large numbers" {

  let large_number : UInt64 = (1UL << 63) - 1UL


  inspect(large_number > (1UL << 32) - 1UL, content="true")


  let result = large_number * 2UL
  inspect(result, content="18446744073709551614")
}
```

Working with Hexadecimal Literals

UInt64 works well with hexadecimal literals for clarity when working with bit patterns:

```

test "UInt64 hexadecimal literals" {

  let value = 0xDEADBEEFUL


  let ad = (value >> 16) & 0xFFUL
  inspect(ad to_byte(), content="b'\\xAD'")


  let bytes = value to_be_bytes()
  inspect(
    bytes,
    content=(
      #|b"\x00\x00\x00\x00\xde\xad\xbe\xef"
    ),
  )
}
```

Queue

Queue is a first in first out (FIFO) data structure, allowing to process their elements in the order they come.

Usage

Create and Clear

You can create a queue manually by using the new or construct it using the from_array.

```

test {
  let _queue : @queue Queue[Int] = @queue new()
  let _queue1 = @queue of([1, 2, 3])

}
```

To clear the queue, you can use the clear method.

```

test {
  let queue = @queue of([1, 2, 3])
  queue clear()
}
```

Length

You can get the length of the queue by using the length method. The is_empty method can be used to check if the queue is empty.

```

test {
  let queue = @queue of([1, 2, 3])
  assert_eq(queue length(), 3)
  assert_eq(queue is_empty(), false)
}
```

Pop and Push

You can add elements to the queue using the push method and remove them using th
e pop method.

```

test {
  let queue = @queue new()
  queue push(1)
  queue push(2)
  assert_eq(queue pop(), Some(1))
  assert_eq(queue pop(), Some(2))
}
```

Peek

You can get the first element of the queue without removing it using the peek me
thod.

```

test {
  let queue = @queue of([1, 2, 3])
  assert_eq(queue peek(), Some(1))
}
```

Traverse

You can traverse the queue using the each method.

```

test {
  let queue = @queue of([1, 2, 3])
  let mut sum = 0
  queue each(x => sum += x)
  assert_eq(sum, 6)
}
```

You can fold the queue using the fold method.

```

test {
  let queue = @queue of([1, 2, 3])
  let sum = queue fold(init=0, (acc, x) => acc + x)
  assert_eq(sum, 6)
}
```

Copy and Transfer

You can copy a queue using the copy method.

```
test {
  let queue = @queue of([1, 2, 3])
  let _queue2 = queue copy()

}
```

Transfer the elements from one queue to another using the transfer method.

```
test {
  let dst : @queue Queue[Int] = @queue new()
  let src : @queue Queue[Int] = @queue of([5, 6, 7, 8])
  src transfer(dst)
}
```

double

This package provides comprehensive support for double-precision floating-point arithmetic, including basic operations, trigonometric functions, exponential and logarithmic functions, as well as utility functions for handling special values
.

Constants and Special Values

The package provides several important constants and special floating-point valu
es:

```
test "special values" {

  inspect(@double infinity, content="Infinity")
  inspect(@double neg_infinity, content="-Infinity")
  inspect(@double not_a_number, content="NaN")


  inspect(@double max_value, content="1.7976931348623157e+308")
  inspect(@double min_value, content="-1.7976931348623157e+308")
  inspect(@double min_positive, content="2.2250738585072014e-308")
}
```

Basic Operations

Basic mathematical operations and rounding functions:

```

test "basic operations" {

  inspect(@double abs(-3.14), content="3.14")


  inspect(@double floor(3.7), content="3")
  inspect(@double ceil(3.2), content="4")
  inspect(@double round(3.5), content="4")
  inspect(@double trunc(3.7), content="3")


  inspect(2.0 pow(3), content="8")


  inspect((-3.14) signum(), content="-1")
  inspect(2.0 signum(), content="1")


  inspect(@double from_int(42), content="42")
}
```

Special Value Testing

Functions for testing special floating-point values and comparing numbers:

```

test "special value testing" {

  inspect(@double not_a_number is_nan(), content="true")
  inspect(@double infinity is_inf(), content="true")
  inspect(@double infinity is_pos_inf(), content="true")
  inspect(@double neg_infinity is_neg_inf(), content="true")


  let relative_tolerance = 1.e-9
  inspect(@double is_close(0.1 + 0.2, 0.3, relative_tolerance~), content
}
```

Binary Representation

Functions for converting doubles to their binary representation:

```

test "binary representation" {
  let num = 1.0



  inspect(
    num.to_be_bytes(),
    content=(
      #|b"\x3f\xf0\x00\x00\x00\x00\x00\x00"
    ),
  )
  inspect(
    num.to_le_bytes(),
    content=(
      #|b"\x00\x00\x00\x00\x00\x00\xf0\x3f"
    ),
  )
}
```

Note: Most methods can be called either as a method (d.to_be_bytes()) or as a package function (@double.to_be_bytes(d)).

Sorted Map

A mutable map backed by an AVL tree that maintains keys in sorted order.

Overview

SortedMap is an ordered map implementation that keeps entries sorted by keys. It provides efficient lookup, insertion, and deletion operations, with stable trav ersal order based on key comparison.

Performance

- **add/set**: O(log n)
- **remove**: O(log n)
- **get/contains**: O(log n)
- **iterate**: O(n)
- **range**: O(log n + k) where k is number of elements in range
- **space complexity**: O(n)

Usage

Create

You can create an empty SortedMap or a SortedMap from other containers.

```
test {
  let _map1 : @sorted_map SortedMap[Int, String] = @sorted_map new()
  let _map2 = @sorted_map from_array([(1, "one"), (2, "two"), (3, "three

}
```

Container Operations

Add a key-value pair to the SortedMap in place.

```
test {
  let map = @sorted_map from_array([(1, "one"), (2, "two")])
  map set(3, "three")
  assert_eq(map size(), 3)
}
```

You can also use the convenient subscript syntax to add or update values:

```
test {
  let map = @sorted_map new()
  map[1] = "one"
  map[2] = "two"
  assert_eq(map size(), 2)
}
```

Remove a key-value pair from the SortedMap in place.

```
test {
  let map = @sorted_map from_array([(1, "one"), (2, "two"), (3, "three")
  map remove(2)
  assert_eq(map size(), 2)
  assert_eq(map contains(2), false)
}
```

Get a value by its key. The return type is Option[V].

```
test {
  let map = @sorted_map from_array([(1, "one"), (2, "two"), (3, "three")
  assert_eq(map get(2), Some("two"))
  assert_eq(map get(4), None)
}
```

Safe access with error handling:

```

test {
  let map = @sorted_map from_array([(1, "one"), (2, "two")])
  let key = 3
  inspect(map get(key), content="None")
}
```

Check if a key exists in the map.

```

test {
  let map = @sorted_map from_array([(1, "one"), (2, "two"), (3, "three")
  assert_eq(map contains(2), true)
  assert_eq(map contains(4), false)
}
```

Iterate over all key-value pairs in the map in sorted key order.

```

test {
  let map = @sorted_map from_array([(3, "three"), (1, "one"), (2, "two")
  let keys = []
  let values = []
  map each((k, v) => {
    keys push(k)
    values push(v)
  })
  assert_eq(keys, [1, 2, 3])
  assert_eq(values, ["one", "two", "three"])
}
```

Iterate with index:

```

test {
  let map = @sorted_map from_array([(3, "three"), (1, "one"), (2, "two")
  let result = []
  map eachi((i, k, v) => result push((i, k, v)))
  assert_eq(result, [(0, 1, "one"), (1, 2, "two"), (2, 3, "three")])
}
```

Get the size of the map.

```

test {
  let map = @sorted_map from_array([(1, "one"), (2, "two"), (3, "three")
  assert_eq(map size(), 3)
}
```

Check if the map is empty.

```

test {
  let map : @sorted_map SortedMap[Int, String] = @sorted_map new()
  assert_eq(map is_empty(), true)
}
```

Clear the map.

```

test {
  let map = @sorted_map from_array([(1, "one"), (2, "two"), (3, "three")
  map clear()
  assert_eq(map is_empty(), true)
}
```

Data Extraction

Get all keys or values from the map.

```

test {
  let map = @sorted_map from_array([(3, "three"), (1, "one"), (2, "two")
  assert_eq(map keys_as_iter() collect(), [1, 2, 3])
  assert_eq(map values_as_iter() collect(), ["one", "two", "three"])
}
```

Convert the map to an array of key-value pairs.

```

test {
  let map = @sorted_map from_array([(3, "three"), (1, "one"), (2, "two")
  assert_eq(map to_array(), [(1, "one"), (2, "two"), (3, "three")])
}
```

Range Operations

Get a subset of the map within a specified range of keys. The range is inclusive
for both bounds [low, high].

```

test {
  let map = @sorted_map from_array([
    (1, "one"),
    (2, "two"),
    (3, "three"),
    (4, "four"),
    (5, "five"),
  ])
  let range_items = []
  map range(2, 4) each((k, v) => range_items push((k, v)))
  assert_eq(range_items, [(2, "two"), (3, "three"), (4, "four")])
}
```

Edge cases for range operations:

- If low > high, returns an empty result
- If low or high are outside the map bounds, returns only pairs within valid bound
s
- The returned iterator preserves the sorted order of keys

```


test {
  let map = @sorted_map from_array([(1, "one"), (2, "two"), (3, "three")
  let range_items = []
  map range(0, 10) each((k, v) => range_items push((k, v)))
  assert_eq(range_items, [(1, "one"), (2, "two"), (3, "three")])


  let empty_range : Array[(Int, String)] = []
  map range(10, 5) each((k, v) => empty_range push((k, v)))
  assert_eq(empty_range, [])
}
```

Iterators

The SortedMap supports several iterator patterns. Create a map from an iterator:

```

test {
  let pairs = [(1, "one"), (2, "two"), (3, "three")] iter()
  let map = @sorted_map from_iter(pairs)
  assert_eq(map size(), 3)
}
```

Use the iter method to get an iterator over key-value pairs:

```

test {
  let map = @sorted_map from_array([(3, "three"), (1, "one"), (2, "two")
  let pairs = map iter() to_array()
  assert_eq(pairs, [(1, "one"), (2, "two"), (3, "three")])
}
```

Use the iter2 method for a more convenient key-value iteration:

```

test {
  let map = @sorted_map from_array([(3, "three"), (1, "one"), (2, "two")
  let transformed = []
  map iter2() each((k, v) => transformed push(k to_string() + ": " + v))
  assert_eq(transformed, ["1: one", "2: two", "3: three"])
}
```

Equality

Maps with the same key-value pairs are considered equal, regardless of the order
 in which elements were added.

```
test {
  let map1 = @sorted_map from_array([(1, "one"), (2, "two")])
  let map2 = @sorted_map from_array([(2, "two"), (1, "one")])
  assert_eq(map1 == map2, true)
}
```

Error Handling Best Practices

When working with keys that might not exist, prefer using pattern matching for s
afety:

```
fn get_score(scores : @sorted_map SortedMap[Int, Int], student_id : Int)
  match scores get(student_id) {
    Some(score) => score
    None =>





        0
  }
}


test "safe_key_access" {

  let scores = @sorted_map from_array([(1001, 85), (1002, 92), (1003, 78


  assert_eq(get_score(scores, 1001), 85)


  assert_eq(get_score(scores, 9999), 0)
}
```

Implementation Notes

The SortedMap is implemented as an AVL tree, a self-balancing binary search tree
. After insertions and deletions, the tree automatically rebalances to maintain
O(log n) search, insertion, and deletion times.

Key properties of the AVL tree implementation:
- Each node stores a balance factor (height difference between left and right sub
  trees)
- The balance factor is maintained between -1 and 1 for all nodes
- Rebalancing is done through tree rotations (single and double rotations)

Comparison with Other Collections

- **@hashmap.T**: Provides O(1) average case lookups but doesn’t maintain order; use when order doesn’t matter
- **@indexmap.T**: Maintains insertion order but not sorted order; use when insertion order matters
- **@sorted_map.SortedMap**: Maintains keys in sorted order; use when you need keys to be sorted

Choose SortedMap when you need:
- Key-value pairs sorted by key
- Efficient range queries
- Ordered traversal guarantees

MoonBit QuickCheck Package

MoonBit QuickCheck package provides property-based testing capabilities by gener
ating random test inputs.

Basic Usage

Generate random values of any type that implements the Arbitrary trait:

```

test "basic generation" {
  let b : Bool = @quickcheck gen()
  inspect(b, content="true")
  let x : Int = @quickcheck gen()
  inspect(x, content="0")


  let sized : Array[Int] = @quickcheck gen(size=5)
  inspect(sized length() <= 5, content="true")
}
```

Multiple Samples

Generate multiple test cases using the samples function:

```

test "multiple samples" {
  let ints : Array[Int] = @quickcheck samples(5)
  inspect(ints, content="[0, 0, 0, -1, -1]")
  let strings : Array[String] = @quickcheck samples(12)
  inspect(
    strings[5:10],
    content=(
      #|["E\b\u{0f} ", "", "K\u{1f}[", "!@", "xvLxb"]
    ),
  )
}
```

Built-in Types

QuickCheck provides Arbitrary implementations for all basic MoonBit types:

```

test "builtin types" {

  let v : (Bool, Char, Byte) = @quickcheck gen()
  inspect(v, content="(true, '#', b'\\x12')")

  let v : (Int, Int64, UInt, UInt64, Float, Double, BigInt) = @quickchec
  inspect(v, content="(0, 0, 0, 0, 0.1430625319480896, 0.330984466952546


  let v : (String, Bytes, Iter[Int]) = @quickcheck gen()
  inspect(
    v,
    content=(
      #|("", b"", [])
    ),
  )
}
```

Custom Types

Implement Arbitrary trait for custom types:

```

struct Point {
  x : Int
  y : Int
} derive(Show)


impl Arbitrary for Point with arbitrary(size, r0) {
  let r1 = r0 split()
  let y = @quickcheck Arbitrary::arbitrary(size, r1)
  { x: @quickcheck Arbitrary::arbitrary(size, r0), y }
}


test "custom type generation" {
  let point : Point = @quickcheck gen()
  inspect(point, content="{x: 0, y: 0}")
  let points : Array[Point] = @quickcheck samples(10)
  inspect(
    points[6:],
    content="[{x: 0, y: 1}, {x: -1, y: -5}, {x: -6, y: -6}, {x: -1, y: 7
  )
}
```

The package is useful for writing property tests that verify code behavior across a wide range of randomly generated inputs.

uint

This package provides functionalities for handling 32-bit unsigned integers in MoonBit. To this end, it includes methods for converting between UInt and other number formats, as well as utilities for byte representation.

Basic Properties

uint provides constants for UInt's value range and default value:

```

test "uint basics" {

  inspect(@uint default(), content="0")


  inspect(@uint max_value, content="4294967295")
  inspect(@uint min_value, content="0")
}
```

Byte Representation

UInt can be converted to bytes in both big-endian and little-endian formats:

```

test "uint byte conversion" {
  let num = 258U


  let be_bytes = num to_be_bytes()
  inspect(
    be_bytes,
    content=(
      #|b"\x00\x00\x01\x02"
    ),
  )


  let le_bytes = num to_le_bytes()
  inspect(
    le_bytes,
    content=(
      #|b"\x02\x01\x00\x00"
    ),
  )
}
```

Converting to Other Number Types

UInt can be converted to Int64 when you need to work with signed 64-bit integers:

```

test "uint type conversion" {
  let num = 42U
  inspect(num to_int64(), content="42")
  let large_num = 4294967295U
  inspect(large_num to_int64(), content="4294967295")
}
```

These conversion functions are also available as methods:

```

test "uint methods" {
  let num = 1000U


  inspect(num to_int64(), content="1000")
  inspect(
    num to_be_bytes(),
    content=(
      #|b"\x00\x00\x03\xe8"
    ),
  )
  inspect(
    num to_le_bytes(),
    content=(
      #|b"\xe8\x03\x00\x00"
    ),
  )
}
```

json

The json package provides comprehensive JSON handling capabilities, including parsing, stringifying, and type-safe conversion between JSON and other MoonBit data types.

Basic JSON Operations

Parsing and Validating JSON

```

test "parse and validate jsons" {

  assert_true(@json valid("{\"key\": 42}"))
  assert_true(@json valid("[1, 2, 3]"))
  assert_true(@json valid("null"))
  assert_true(@json valid("false"))


  let json = @json parse("{\"key\": 42}") catch {
    (_ : @json ParseError) => panic()

  }


  inspect(
    json stringify(indent=2),
    content={
      let output =
        #|{
        #|  "key": 42
        #|}
      output
    },
  )
}
```

Object Navigation

```

test "json object navigation" {
  let json = @json parse(
    "{\"string\":\"hello\",\"number\":42,\"array\":[1,2,3]}",
  )


  let string_opt = json value("string") unwrap() as_string()
  inspect(
    string_opt,
    content=(
      #|Some("hello")
    ),
  )


  let number_opt = json value("number") unwrap() as_number()
  inspect(number_opt, content="Some(42)")


  let array_opt = json value("array") unwrap() as_array()
  inspect(array_opt, content="Some([Number(1), Number(2), Number(3)])")


  inspect(json value("missing"), content="None")
}
```

Array Navigation

```

test "json array navigation" {
  let array = @json parse("[1,2,3,4,5]")


  let first = array item(0)
  inspect(first, content="Some(Number(1))")


  let missing = array item(10)
  inspect(missing, content="None")


  let values = array as_array() unwrap()
  inspect(
    values iter(),
    content="[Number(1), Number(2), Number(3), Number(4), Number(5)]",
  )
}
```

Type-Safe JSON Conversion

From JSON to Native Types

```

test "json decode" {

  let json_number = (42 : Json)
  let number : Int = @json from_json(json_number)
  inspect(number, content="42")


  let json_array = ([1, 2, 3] : Json)
  let array : Array[Int] = @json from_json(json_array)
  inspect(array, content="[1, 2, 3]")


  let json_map = ({ "a": 1, "b": 2 } : Json)
  let map : Map[String, Int] = @json from_json(json_map)
  inspect(
    map,
    content=(
      #|{"a": 1, "b": 2}
    ),
  )
}
```

Error Handling with JSON Path

```

test "json path" {

  try {
    let _arr : Array[Int] = @json from_json(([42, "not a number", 49] :
    panic()
  } catch {
    @json JsonDecodeError((path, msg)) => {
      inspect(path, content="$[1]")
      inspect(msg, content="Int::from_json: expected number")
    }
  }
}
```

JSON-based Snapshot Testing

@json.inspect() can be used as an alternative to inspect() when a value’s To
Json implementation is considered a better debugging representation than its Sho
w implementation. This is particularly true for deeply-nested data structures.

```

test "json inspection" {
  let null = null


  let json_value : Json = { "key": "value", "numbers": [1, 2, 3] }
  @json inspect(json_value, content={ "key": "value", "numbers": [1, 2,


  let json_special = { "null": null, "bool": true }
  @json inspect(json_special, content={ "null": null, "bool": true })
}
```

Env Package Documentation

This package provides utilities for interacting with the runtime environment, including access to command line arguments, current time, and working directory information.

Command Line Arguments

Access command line arguments passed to your program:

```

test "command line arguments" {
  let arguments = @env args()



  inspect(arguments length() >= 0, content="true")


  fn process_args(args : Array[String]) -> String {
    if args length() == 0 {
      "No arguments provided"
    } else {
      "First argument: " + args[0]
    }
  }

  let result = process_args(arguments)
  inspect(result length() > 0, content="true")
}
```

Current Time

Get the current time in milliseconds since Unix epoch:

```

test "current time" {
  let timestamp = @env now()


  let year_2020_ms = 1577836800000UL
  inspect(timestamp > year_2020_ms, content="true")


  fn format_timestamp(ts : UInt64) -> String {
    "Timestamp: " + ts to_string()
  }

  let formatted = format_timestamp(timestamp)
  inspect(formatted length() > 10, content="true")
}
```

Working Directory

Get the current working directory:

```

test "working directory" {
  let cwd = @env current_dir()
  match cwd {
    Some(path) => {

      inspect(path length() > 0, content="true")
      inspect(path length() > 1, content="true")
    }
    None =>

      inspect(true, content="true")
  }
}
```

Practical Usage Examples

Command Line Tool Pattern

```
test "command line tool pattern" {
  fn parse_command(args : Array[String]) -> Result[String, String] {
    if args.length() < 2 {
      Err("Usage: program <command> [args...]")
    } else {
      match args[1] {
        "help" => Ok("Showing help information")
        "version" => Ok("Version 1.0.0")
        "status" => Ok("System is running")
        cmd => Err("Unknown command: " + cmd)
      }
    }
  }


  let test_args = ["program", "help"]
  let result = parse_command(test_args)
  inspect(result, content="Ok(\"Showing help information\")")
  let invalid_result = parse_command(["program", "invalid"])
  match invalid_result {
    Ok(_) => inspect(false, content="true")
    Err(msg) => inspect(msg.length() > 10, content="true")
  }
}
```

Configuration Loading

```
test "configuration loading" {
  fn load_config_path() -> String {
    match @env.current_dir() {
      Some(cwd) => cwd + "/config.json"
      None => "./config.json"
    }
  }

  let config_path = load_config_path()
  inspect(config_path.length() > 10, content="true")
}
```

Logging with Timestamps

```
test "logging with timestamps" {
  fn log_message(level : String, message : String) -> String {
    let timestamp = @env.now()
    "[" + timestamp.to_string() + "] " + level + ": " + message
  }

  let log_entry = log_message("INFO", "Application started")
  inspect(log_entry.length() > 20, content="true")
  inspect(log_entry.length() > 10, content="true")
}
```

File Path Operations

```

test "file path operations" {
  fn resolve_relative_path(relative : String) -> String {
    match @env current_dir() {
      Some(base) => base + "/" + relative
      None => relative
    }
  }

  let resolved = resolve_relative_path("data/input.txt")
  inspect(resolved length() > 10, content="true")
}
```

Platform Differences

The env package behaves differently across platforms:

JavaScript Environment

- args() returns arguments from the JavaScript environment
- @env.now() uses Date.@env.now()
- @env.current_dir() may return None in browser environments

WebAssembly Environment

- args() behavior depends on the WASM host
- @env.now() provides millisecond precision timing
- @env.current_dir() availability depends on host capabilities

Native Environment

- args() returns actual command line arguments
- @env.now() provides system time
- @env.current_dir() uses system calls to get working directory

Error Handling

Handle cases where environment information is unavailable:

```
test "error handling" {
  fn safe_get_cwd() -> String {
    match @env current_dir() {
      Some(path) => path
      None =>

        "."
    }
  }

  let safe_cwd = safe_get_cwd()
  inspect(safe_cwd length() > 0, content="true")
  fn validate_args(
    args : Array[String],
    min_count : Int,
  ) -> Result[Unit, String] {
    if args length() < min_count {
      Err("Insufficient arguments: expected at least " + min_count to_st
    } else {
      Ok(())
    }
  }

  let validation = validate_args(["prog"], 2)
  match validation {
    Ok(_) => inspect(false, content="true")
    Err(msg) => inspect(msg length() > 10, content="true")
  }
}
```

Best Practices

1. Handle Missing Environment Data Gracefully

```
test "graceful handling" {
  fn get_work_dir() -> String {
    match @env current_dir() {
      Some(dir) => dir
      None => "~"
    }
  }

  let work_dir = get_work_dir()
  inspect(work_dir length() > 0, content="true")
}
```

2. Validate Command Line Arguments

```

test "argument validation" {
  fn validate_and_parse_args(
    args : Array[String],
  ) -> Result[(String, Array[String]), String] {
    if args length() == 0 {
      Err("No program name available")
    } else if args length() == 1 {
      Ok((args[0], []))
    } else {
      let program = args[0]
      let arguments = Array::new()
      for i in 1..<args length() {
        arguments push(args[i])
      }
      Ok((program, arguments))
    }
  }

  let test_result = validate_and_parse_args(["myprogram", "arg1", "arg2"
  match test_result {
    Ok((prog, args)) => {
      inspect(prog, content="myprogram")
      inspect(args length(), content="2")
    }
    Err(_) => inspect(false, content="true")
  }
}
```

3. Use Timestamps for Unique Identifiers

```

test "unique identifiers" {
  fn generate_unique_id(prefix : String) -> String {
    prefix + "_" + @env now() to_string()
  }

  let id1 = generate_unique_id("task")
  let id2 = generate_unique_id("task")
  inspect(id1 length() > 10, content="true")
  inspect(id2 length() > 10, content="true")

}
```

Common Use Cases

- **Command Line Tools**: Parse arguments and provide help/usage information
- **Configuration Management**: Load config files relative to current directory
- **Logging Systems**: Add timestamps to log entries
- **File Processing**: Resolve relative file paths
- **Debugging**: Include environment information in error reports
- **Build Tools**: Determine working directory for relative path operations

Performance Considerations

- args() is typically called once at program startup
- @env.now() is lightweight but avoid calling in tight loops if high precision isn’t needed
- @env.current_dir() may involve system calls, so cache the result if used frequently
- Environment functions are generally fast but platform-dependent

The env package provides essential runtime environment access for building robust MoonBit applications that interact with their execution environment.