Cairo Language Guide

Table of Contents

Beginners Guide

If you are a total beginner to this, start here!

1. Read the official Cairo Book Cairo Book
2. Practice with Starklings Starklings
3. Read cairo core library features Cairo Core Library
4. Starknet By Example for Smart Contracts and other stuff Starknet By Example
5. Practice more technical Cairo problems Node Guardians

Immutability

Variables in Cairo are immutable by default.

```
fn main() {
  let x = 5;
  println!("Value of x is: {}", x);
  x = 6; // This won't compile.
  println!("Value of x is: {}", x);
}
```

> However we are allowed to change the value bound to x from 5 to 6 when mut is used.

```
fn main() {
  let mut x = 5;
  println!("The value of x is: {}", x); // Prints 5
  x = 6;
  println!("The value of x is: {}", x); // Prints 6
}
```

Constants

Constants are always immutable (you cannot use mut with constants).

```
const ONE_HOUR_IN_SECONDS: u32 = 3600;
```

Shadowing

Refers to the declaration of a new variable with the same name as a previous variable.

```
fn main() {
  let x = 10;
  let x = x + 1;
  {
      let x = x * 2;
```

```
        println!("Inner scope x value is: {}", x); // Value is 22
    }
    println!("Outer scope x value is: {}", x); // Value is 11
}
```

Datatypes: felt252

- Represents a field element with the keyword felt252.
- A field element is an integer in the range -X < felt < X, where X = 2^{251} + 17* 2^{192} + 1
- Felt252 is the basis for creating all types in the core library.
- Felt252 can store both integers and strings.

```
let y: felt252 = 1;
let x: felt252 = 5;
let y: felt252 = 'Hello World';
```

Datatypes: integers

- An integer is a number without a fractional component.
- Cairo provides a range of integer types, including u8, u16, u32, u64, u128, u256 and usize.
- Cairo also provides support for signed integers, starting with the prefix i

```
let y = 2;
let x:u8 = 2; // To specify u8, u16, u32 and other types
```

Datatypes: boolean

Has two possible values: true and false

```
let true = true;
let false: bool = false; // Explicit type annotation
```

Datatypes: short string

- It is possible to store characters called as "short string" inside felt252.
- Max length of 31 chars.

```
let my_first_char = 'F';
let my_first_string = 'Hello world';
```

Datatypes: ByteArrays

Strings can be represented with the following ByteArray struct:

```
#[derive(Drop, Clone, PartialEq, Serde, Default)]
struct ByteArray {
    data: Array<bytes31>,
    pending_word: felt252,
    pending_word_len: usize,
}
```

```
let my_string: ByteArray = "this is a string which has more than 31 characters";
```

Control Flow: match

Enables you to evaluate a value against a sequence of patterns and subsequently execute code depending on the matched pattern.

```
enum Currency {
  Dolar,
  Pound,
  Euro,
  Dinar,
}

fn currency_amount(currency: Currency) -> felt252 {
  match currency {
      Currency::Dolar => 10,
      Currency::Pound => 50,
      Currency::Euro => 100,
      Currency::Dinar => 250,
  }
}
```

Control Flow: if/else statements

- An if expression permits branching in your code based on conditions.
- Alternatively, we may include an else expression, providing the program with an alternative block of code to execute if the condition evaluates to false.

```
let number = 10;

if number == 12 {
  println!("number is 12");
}
else if number == 10 {
  println!("number is 10");
}
```

```
else if number - 2 == 1 {
  println!("number minus 2 is 1");
}
else {
  println!("number not found");
}
```

Control Flow: loops

Executes a block code until we explicitly indicate to stop. The break stops the loop, and continue goes to the next iteration of the code.

```
let mut i: usize = 0;

loop {
    if i > 10 {
      break;
    }
    if i == 7 {
      i += 1;
      continue;
    }
    println!("i = {}", i);
    i += 1;
} // This program will not print the value of i when it is equal to 7
```

Control Flow: while

while works by evaluating a condition within a loop

```
let mut number = 10;

while number != 0 {
  println!("{number}!");
  number -= 1;
};
```

Functions

- Defined by fn followed by a set of parenthesis and curly brackets to delimit its body.
- They can accept defined parameters by declaring the type of each argument (function signatures) using parameter_name: value.

```
fn simple_function(x: u16, y: u16) {}

fn main() {
  let first_param = 3;
  let second_param = 4;
```

```
    simple_function(x: first_param, y: second_param);

    let x = 1;
    let y = 2;

    simple_function(:x, :y) // This is also valid
}
```

> Functions can return values using return < value > or returning the last expression implicitly.

```
fn return_ten() -> u32 { // Type is specified.
  10
}
```

Advanced Types: tuples

A tuple is created by enclosing a comma-separated list of values within parentheses.

```
let tup1: (u32, u64, bool) = (10, 20, true);
let tup2 = (500, 6, true);
let (x, y, z) = tup2;
if y == 6 {
  println!("y is 6!"); // The program prints: y is 6
}
```

> A unit type is a type which has only one value (). It is represented by a tuple with no elements.

Advanced Types: structs

Collection of custom user types defined by named fields called members.

```
struct Person {
  age: u8,
  height: u8,
  name: felt252, // String
}
```

Advanced Types: enums

A custom data type comprising a predetermined collection of named values, referred to as variants.

```
enum Colors {
  Red,
```

```
    Blue,
    Pink,
    Green,
}
```

Advanced Types: arrays

You can generate and utilize array methods through the ArrayTrait trait in the core library.

- Creating an array and appending 3 elements:

```
let mut a = ArrayTrait::new();
a.append(0);
a.append(1);
a.append(2);
```

- You can specify the expected type of items within the array or explicitly define the variable's type:

```
let mut arr = ArrayTrait::<u128>::new();
let mut arr:Array<u128> = ArrayTrait::new();
```

- Adding & deleting elements:

```
arr.append(1)
// You can only remove elements from the front of an array
let first_value = arr.pop_front().unwrap();
```

- Reading elements of an array

```
// get() returns an Option<Box<@T>>
arr.get(index_to_access)
let mut a = ArrayTrait::new();
a.append(0);
a.append(1);

// at() returns a snapshot to the element at the specified index
let first = *a.at(0);
let second = *a.at(1);
```

Advanced Types: dictionaries

The Felt252Dict<T> data type embodies a set of unique key-value pairs, where each key is associated with a corresponding value. Core functionality includes:

- insert(felt252, T) -> () to write values to a dictionary instance.

- get(felt252) -> T to read values from it.

```
let mut balances: Felt252Dict<u64> = Default::default();
balances.insert('Marco', 100);
balances.insert('Luis', 200);

let marco_balance = balances.get('Marco');
assert!(marco_balance == 100, "Balance is not 100");

let luis_balance = balances.get('Luis');
assert!(luis_balance == 200, "Balance is not 200");
```

Traits

Trait definitions serve as a means to assemble method signatures, defining a collection of behaviors essential for achieving a specific purpose.

```
trait Feed {
   fn get_feed(self: @BlogPost) -> ByteArray;
}
```

> You can write implementations directly without defining the corresponding trait. This is made possible by using the #[generate_trait] attribute within the implementation.

```
struct Square {
  height: u64,
}

#[generate_trait]
impl SquareGeometry of SquareGeometryTrait {
  fn area(self: Square) -> u64 {
  self.height * self.height
  }
}
```

Ownership

Each value within Cairo is exclusively owned by one owner at any given time.

```
fn my_function(arr: Array<u128>) {
    // my_function takes ownership of the array.
    // when this function returns, arr is dropped.
}
fn main() {
  // as the creator of arr, the main function owns the array
  let arr = ArrayTrait::<u128>::new();
  my_function(arr); // moves ownership of the array to function call
}
```

Snapshots

Snapshots offer immutable object instances without assuming ownership. To create a snapshot of a value x use @x.

```
// Receives an array snapshot
fn some_function(data: @Array<u32>) -> u32 {
// data.append(5_u32); This will fail, as data is read-only
}
```

Mutable References

Mutable values passed to a function returning ownership to the calling context using the ref modifier.

```
#[derive(Copy, Drop)]
struct Rectangle {
  height: u64,
  width: u64,
}

fn main() {
  let mut rec = Rectangle { height: 3, width: 10 };
  flip(ref rec);
  /// height: 10, width: 3
  println!("height: {}, width: {}", rec.height, rec.width);
}

fn flip(ref rec: Rectangle) {
  let temp = rec.height;
  rec.height = rec.width;
  rec.width = temp;
}
```

Testing Contracts/Code

The #[test] annotation indicates that the function is a test function.

```
#[test]
fn testing_function() {
  let result = 5 * 2;
  assert!(result == 10, "result is not 10");
}
```

assert! macro

Enables checking if a condition holds true and panicking otherwise, along with providing a specified panic string that can be formatted.

```
let var1 = 10;
let var2 = 20;
assert!(var1 != var2, "should not be equal");
assert!(var1 != var2, "{},{} should not be equal", var1, var2);
```

Testing Equality

Compare two arguments for equality or inequality with assert_eq!

```
fn add_two(a: u32) -> u32 {
  a + 2
}

#[test]
fn it_adds_two() {
  assert_eq!(4, add_two(2));
}
```

Benchmarking gas usage

Use the following pattern to benchmark gas usage:

```
let initial = testing::get_available_gas();
gas::withdraw_gas().unwrap();
// Code we want to bench.
println!("{}\n", initial - testing::get_available_gas());
```

Error handling: panic! macro

Panic results in the termination of the program. The panic macro takes the panic error as its input.

```
panic!("Panicking!, but at least I'm not limited to 31 characters anymore like a
```

Error handling: nopanic notation

Use the nopanic notation to indicate that a function will never panic

```
fn function_never_panic() -> felt252 nopanic {
  1
}
```

Comments

Leave comments in your code using the following syntax:

```
fn my_function() -> string {
  // Beginning of the function
  5 + 11 // returns the sum of 5 and 11
}
```

Created on May 12, 2025 by Marco Araya - @coxmar_devCR

tags: `Cairo` `Documentation`