Cairo Syntax Cheatsheet

A list of useful chunk codes for your reference!

Data Structures recap
Immutability
Constants
Shadowing
Data type: Felts
Data type: Integers
Numeric operations
Data Type: Boolean
Data Type: Short String
Data Type: ByteArrays

Control Flow
Statements and Expressions
Match Control Flow
Control Flow: if/else statements
Control Flow: loops
Control Flow: while

Functions and common collections
Functions
Arrays
Dictionaries

Contracts in Cairo
Contracts
Storage
Storage Access
Interfaces
Implementations
Constructors
Events

Testing in Cairo
Testing Contracts
assert! macro
Comments
Testing Equality
Benchmarking gas usage for an operation
Error handling: panic! macro
Error handling: nopanic notation

Starknet Foundry
Deploying a contract
Generating a new address
Impersonating a User
Changing the block timestamp for a contract

Advanced Types
Data type: Tuples
Structs
Enums
Traits

What is Ownership?
Clone and Copy
Ownership
Snapshots
Mutable references
Spans

Components & System Calls
Components
System Calls

Data Structures recap

Immutability

(back to index...)

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

(back to index...)

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

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

Shadowing

(back to index...)

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

- Note: Transformations occur on the value but the variable remains immutable after those transformations complete.

```
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
}
```

Data type: Felts

(back to index...)

- 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 = 'pppp';
```

Data type: Integers

(back to index...)

- 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, and other types
```

Numeric operations

(back to index...)

Integers support most operators (addition, subtraction, multiplication, division, and remainder) and come with overflow detection, supporting the following:

```
fn main() {
    // addition
    let sum = 10_u128 + 10_u128;

    // subtraction
    let difference = 95_u128 - 94_u128;

    // multiplication
    let product = 4_u128 * 5_u128;

    // division
    let quotient = 56_u128 / 32_u128; // result is 1
    let quotient = 32_u128 / 16_u128; // result is 2

    // remainder
    let remainder = 42_u128 % 5_u128; // result is 2
}
```

Data Type: Boolean

(back to index...)

Has two possible values: `true` and `false`.

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

Data Type: Short String

(back to index...)

- 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';
```

Data Type: ByteArrays

(back to index...)

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

Statements and Expressions

(back to index...)

- Statements are commands that execute actions without returning a value.
- Expressions yield a final value upon evaluation.

```
let y = 6; // This is a statement since does not return a value.
let y = {
    let x = 3;
    x + 1
}; // This is a expression. It evaluates to 4.
```

Match Control Flow

(back to index...)

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

(back to index...)

- 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

(back to index...)

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

(back to index...)

`while` works by evaluating a condition within a loop.

```
let mut number = 10;

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

Functions and common collections

Functions

(back to index...)

- 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
}
```

Arrays

(back to index...)

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);
```

- The array ! macro - values already known at compilation time

```
let arr = array![1, 2, 3, 4, 5];
```

Dictionaries

(back to index...)

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('Alice', 100);
balances.insert('Bob', 200);
let alice_balance = balances.get('Alice');
assert!(alice_balance == 100, "Balance is not 100");
let bob_balance = balances.get('Bob');
assert!(bob_balance == 200, "Balance is not 200");
```

Contracts in Cairo

Contracts

(back to index...)

Defining a new contract:

```
#[starknet::contract]
mod Contract {

}
```

Storage

(back to index...)

A contract MUST have a storage or it won't compile, storage is defined inside a `struct`:

```
#[starknet::contract]
mod Contract {

    #[storage]
    struct Storage {
        token_supply: felt252,
        decimals: u8
    }
}
```

Storage Access

(back to index...)

We can use `read` and `write` functions to access the contract's storage:

```
// Read
let current_balance = self.balance.read();
// Write
self.balance.write(current_balance + amount)
```

Interfaces

(back to index...)

- Defines the contract functionality.
- Helpful when you want to interact with another contract.
- Defined by adding a `#[starknet::interface]` attribute to a `trait`:

```
#[starknet::interface]
trait IContract<TContractState> {
   fn increase_token_supply(ref self: TContractState, amount: felt252);
   fn increase_decimals(ref self: TContractState, amount: u8);
   fn get_token_supply(self: @TContractState) -> felt252;
}
```

Implementations

(back to index...)

Implement a trait inside a contract, we use the `impl` keyword, with a `abi(embed_v0)` attribute:

```
mod Contract {

    #[abi(embed_v0)]
    impl Contract of super::IContract<ContractState> {
        fn increase_token_supply(ref self: ContractState, amount: felt252)
{ ... }
        fn increase_decimals(ref self: ContractState, amount: u8) { ... }
        fn get_token_supply(self: @ContractState) -> felt252 { ... }
    }
}
```

Constructors

(back to index...)

Code that will be executed once upon contract deployment, defined as follows:

```
mod Contract {

    #[constructor]
    fn constructor(ref self: ContractState, initial_token_supply: felt252,
initial_decimals: u8) {
        self.token_supply.write(initial_token_supply);
        self.decimals.write(initial_decimals);
    }
}
```

Events

(back to index...)

- Can be emitted to indicated state changes
- Used mainly by the frontend

You can define an event using the following syntax:

```
#[event]
#[derive(Drop, starknet::Event)]
enum Event {
    TokenSupplyIncreased: TokenSupplyIncreased,
    DecimalsIncreased: DecimalsIncreased
}

#[derive(Drop, starknet::Event)]
struct TokenSupplyIncreased {
    amount: felt252
}

#[derive(Drop, starknet::Event)]
struct DecimalsIncreased {
    amount: u8
}
```

Emitting an event:

```
fn increase_token_supply(ref self: ContractState, amount: felt252) {
    ...
    self.emit(TokenSupplyIncreased { amount });
}
```

Testing in Cairo

Testing Contracts

(back to index...)

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

(back to index...)

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);
```

Comments

(back to index...)

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
}
```

Testing Equality

(back to index...)

Compare two arguments for equality or inequality with assert_eq! or assert_ne!.

```
fn add_two(a: u32) -> u32 {
    a + 2
}
#[test]
fn it_adds_two() {
    assert_eq!(4, add_two(2));
}
```

Benchmarking gas usage for an operation

(back to index...)

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

(back to index...)

- Panic results in the termination of the program (after destructing all non-droppable outstanding types). - 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 short string")
```

Error handling: nopanic notation

(back to index...)

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

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

Starknet Foundry

Deploying a contract

(back to index...)

1. Decalre the contract
2. Deploy the contract and get it's deployed address
3. Create a Dispatcher so we can interact with the contract functions.

```
    // Declare the contract
    let contract = declare("ContractName").unwrap();

    // Deploying with an empty array since the constructor doesn't receive
anything
    // using .unwrap to get the contract deplyed address
    let (contract_address, _) = contract.deploy(@array![]).unwrap();

    // Creating a Dispatcher (we need to import the dispatcher interface)
    let dispatcher = IContractNameDispatcher { contract_address };
```

Generating a New Address

(back to index...)

We are essentially converting an integer to an address type 😃

```
    let user1: ContractAddress = 1.try_into().unwrap();
    let user2: ContractAddress = 2.try_into().unwrap();
    let user3: ContractAddress = 3.try_into().unwrap();
```

Impersonating a User

(back to index...)

```
    let user1: ContractAddress = 1.try_into().unwrap();
    start_prank(CheatTarget::One(contract_dispatcher), user1);
    // Do something...
    stop_prank(CheatTarget::One(contract_dispatcher));
```

Changing the block timestamp for a contract

(back to index...)

We change the block timestamp to 1337 for the `contract_dispatcher`.

```
    start_warp(CheatTarget::One(contract_dispatcher), 1337);
    // Do something...
    stop_warp(CheatTarget::One(contract_dispatcher));
```

Advanced types

Data type: Tuples

(back to index...)

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.

Structs

(back to index...)

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

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

Enums

(back to index...)

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

```
enum Colors {
    Red,
    Blue,
    Pink,
    Green,
}
```

- Enums can also retain data linked with each variant.

```
enum Message {
    Alfa,
    Bravo: felt252,
    Delta: (u128, u128),
}
```

Traits

(back to index...)

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
    }
}
```

What is Ownership?

Clone and Copy

(back to index...)

Both are traits:

- Clone specifies how a type instance's data can be duplicated and assigned a new owner.

```
use array::ArrayTrait;
use clone::Clone;
use array::ArrayTCloneImpl;

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.clone()); // moves ownership of a clone of arr to
function call

    my_function(arr); // compiles because previously the array was
duplicated

    // my_function(arr); <- fails to compile, as main doesn't own the
array anymore
}
```

- Copy requires `Clone` and denotes that a type is eligible for bitwise copy.

```
use clone::Clone;

#[derive(Copy, Clone, Drop)]
struct MyVector {
    x: u32,
    y: u32,
}

fn main() {
    let vector = MyVector { x: 1, y: 0 };
    let vector_copy = vector;

    // now vector_copy is a copy of vector, vector is still accesible
}
```

Ownership

(back to index...)

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

(back to index...)

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

(back to index...)

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;
}
```

Spans

(back to index...)

Represents a snapshot of an Array.

```
use array::ArrayTrait;
use array::SpanTrait;

// Receives a Span
fn sum_span(data: Span<u32>) -> u32 {
  *data[0] + *data[1]
}

fn main() -> u32 {
    let mut data: Array<u32> = ArrayTrait::new();
    data.append(1_u32);
    data.append(2_u32);
    sum_span(data.span()) // Using a Span
}
```

Components & System Calls

Components

(back to index...)

A component is very similar to a contract. It can contain:

- Storage variables
- Events
- External and internal functions
-

Unlike a contract, a component cannot be deployed on its own. The component's code becomes part of the contract it's embedded to.

Example of an Ownable Component:

```
#[starknet::interface]
trait IOwnable<TContractState> {
    fn owner(self: @TContractState) -> ContractAddress;
    fn transfer_ownership(ref self: TContractState, new_owner:
ContractAddress);
    fn renounce_ownership(ref self: TContractState);
}
```

The component is defined as:

```
#[starknet::component]
mod ownable_component {
    use starknet::ContractAddress;
    use starknet::get_caller_address;
    use super::Errors;
    use core::zeroable::Zeroable;

    #[storage]
    struct Storage {
        owner: ContractAddress
    }

    #[event]
    #[derive(Drop, starknet::Event)]
    enum Event {
        OwnershipTransferred: OwnershipTransferred
    }

    #[derive(Drop, starknet::Event)]
    struct OwnershipTransferred {
        previous_owner: ContractAddress,
        new_owner: ContractAddress,
    }

    #[embeddable_as(Ownable)]
    impl OwnableImpl<
        TContractState, +HasComponent<TContractState>
    > of super::IOwnable<ComponentState<TContractState>> {
        fn owner(self: @ComponentState<TContractState>) -> ContractAddress
{
            self.owner.read()
        }

        fn transfer_ownership(
            ref self: ComponentState<TContractState>, new_owner:
ContractAddress
        ) {
            assert(!new_owner.is_zero(), Errors::ZERO_ADDRESS_OWNER);
            self.assert_only_owner();
            self._transfer_ownership(new_owner);
        }
        fn renounce_ownership(ref self: ComponentState<TContractState>) {
            self.assert_only_owner();
            self._transfer_ownership(Zeroable::zero());
        }
    }
    #[generate_trait]
    impl InternalImpl<
        TContractState, +HasComponent<TContractState>
    > of InternalTrait<TContractState> {
        fn initializer(ref self: ComponentState<TContractState>, owner:
ContractAddress) {
            self._transfer_ownership(owner);
        }

        fn assert_only_owner(self: @ComponentState<TContractState>) {
            let owner: ContractAddress = self.owner.read();
            let caller: ContractAddress = get_caller_address();
            assert(!caller.is_zero(), Errors::ZERO_ADDRESS_CALLER);
            assert(caller == owner, Errors::NOT_OWNER);
        }

        fn _transfer_ownership(
            ref self: ComponentState<TContractState>, new_owner:
ContractAddress
        ) {
            let previous_owner: ContractAddress = self.owner.read();
            self.owner.write(new_owner);
            self
                .emit(
                    OwnershipTransferred { previous_owner: previous_owner,
new_owner: new_owner }
                );
        }
    }
}
```

Migrating a contract to a Component:

- Add the #[starknet::component] attribute to the module.
- Add the #[embeddable_as(name)] attribute to the impl block that will be embedded in another contract.
- Add generic parameters to the impl block:
- Add TContractState as a generic parameter.
- Add +HasComponent as an impl restriction.
- Change the type of the self argument in the functions inside the impl block to ComponentState instead of ContractState.

Using Components inside a contract (for example embed the Ownable component defined above):

```
#[starknet::contract]
mod OwnableCounter {
    use listing_01_ownable::component::ownable_component;

    component!(path: ownable_component, storage: ownable, event:
OwnableEvent);

    #[abi(embed_v0)]
    impl OwnableImpl = ownable_component::Ownable<ContractState>;

    impl OwnableInternalImpl =
ownable_component::InternalImpl<ContractState>;

    #[storage]
    struct Storage {
        counter: u128,
        #[substorage(v0)]
        ownable: ownable_component::Storage
    }


    #[event]
    #[derive(Drop, starknet::Event)]
    enum Event {
        OwnableEvent: ownable_component::Event
    }


    #[abi(embed_v0)]
    fn foo(ref self: ContractState) {
        self.ownable.assert_only_owner();
        self.counter.write(self.counter.read() + 1);
    }
}
```

Interact with the component's functions externally:

```
#[starknet::interface]
trait IOwnable<TContractState> {
    fn owner(self: @TContractState) -> ContractAddress;
    fn transfer_ownership(ref self: TContractState, new_owner:
ContractAddress);
    fn renounce_ownership(ref self: TContractState);
}
```

System Calls

(back to index...)

System calls allow a contract to demand services from the Starknet OS. All the available syscalls are in the official documentation

Example: `call_contract` -> Calls a given contract. This system call expects the address of the called contract, a selector for a function within that contract, and call arguments.

```
extern fn call_contract_syscall(
    address: ContractAddress, entry_point_selector: felt252, calldata:
Span<felt252>
) -> SyscallResult<Span<felt252>> implicits(GasBuiltin, System) nopanic;
```