# Functions
Functions are prevalent in Cairo code. You've already seen one of the most important functions in the language: the `main` function, which is the entry point of many programs. You've also seen the `fn` keyword, which allows you to declare new functions.
Cairo code uses _snake case_ as the conventional style for function and variable names, in which all letters are lowercase and underscores separate words.
Here's a program that contains an example function definition:
```cairo
{{#include ../listings/ch02-common-programming-concepts/no_listing_15_functions/src/lib.cairo}}
```
We define a function in Cairo by entering `fn` followed by a function name and a set of parentheses. The curly brackets tell the compiler where the function body begins and ends.
We can call any function we've defined by entering its name followed by a set of parentheses. Because `another_function` is defined in the program, it can be called from inside the `main` function. Note that we defined `another_function` _before_ the `main` function in the source code; we could have defined it after as well. Cairo doesn't care where you define your functions, only that they're defined somewhere in a scope that can be seen by the caller.
Let's start a new project with Scarb named _functions_ to explore functions further. Place the `another_function` example in _src/lib.cairo_ and run it. You should see the following output:
```shell
{{#include ../listings/ch02-common-programming-concepts/no_listing_15_functions/output.txt}}
```
The lines execute in the order in which they appear in the `main` function. First the `Hello, world!` message prints, and then `another_function` is called and its message is printed.
## Parameters
We can define functions to have _parameters_, which are special variables that are part of a function's signature. When a function has parameters, you can provide it with concrete values for those parameters. Technically, the concrete values are called _arguments_, but in casual conversation, people tend to use the words _parameter_ and _argument_ interchangeably for either the variables in a function's definition or the concrete values passed in when you call a function.
In this version of `another_function` we add a parameter:
```cairo
{{#include ../listings/ch02-common-programming-concepts/no_listing_16_single_param/src/lib.cairo}}
```
Try running this program; you should get the following output:
```shell
```

{{#include ../listings/ch02-common-programming-concepts/no_listing_16_single_param/output.txt}}
~~~
The declaration of `another_function` has one parameter named `x`. The type of `x` is specified as `felt252`. When we pass `5` in to `another_function`, the `println!` macro puts `5` where the pair of curly brackets containing `x` was in the format string.
In function signatures, you _must_ declare the type of each parameter. This is a deliberate decision in Cairo's design: requiring type annotations in function definitions means the compiler almost never needs you to use them elsewhere in the code to figure out what type you mean. The compiler is also able to give more helpful error messages if it knows what types the function expects.
When defining multiple parameters, separate the parameter declarations with commas, like this:
```cairo
{{#include ../listings/ch02-common-programming-concepts/no_listing_17_multiple_params/src/lib.cairo}}
~~~
This example creates a function named `print_labeled_measurement` with two parameters. The first parameter is named `value` and is a `u128`. The second is named `unit_label` and is of type `ByteArray` - Cairo's internal type to represent string literals. The function then prints text containing both the `value` and the `unit_label`.
Let's try running this code. Replace the program currently in your _functions_ project's _src/lib.cairo_ file with the preceding example and run it using `scarb cairo-run`:
```shell
{{#include ../listings/ch02-common-programming-concepts/no_listing_17_multiple_params/output.txt}}
```
Because we called the function with `5` as the value for value and `"h"` as the value for `unit_label`, the program output contains those values.
### Named Parameters
In Cairo, named parameters allow you to specify the names of arguments when you call a function. This makes the function calls more readable and self-descriptive.
If you want to use named parameters, you need to specify the name of the parameter and the value you want to pass to it. The syntax is `parameter_name: value`. If you pass a variable that has the same name as the parameter, you can simply write `:parameter_name` instead of `parameter_name: variable_name`.
Here is an example:
```cairo
{{#include ../listings/ch02-common-programming-concepts/no_listing_18_named_parameters/src/lib.cairo}}
```
## Statements and Expressions
Function bodies are made up of a series of statements optionally ending in an expression. So far, the functions we've covered haven't included an ending

expression, but you have seen an expression as part of a statement. Because
Cairo is an expression-based language, this is an important distinction to
understand. Other languages don’t have the same distinctions, so let’s look at
what statements and expressions are and how their differences affect the bodies
of functions.
- **Statements** are instructions that perform some action and do not return
  a value.
- **Expressions** evaluate to a resultant value. Let’s look at some examples.
We’ve actually already used statements and expressions. Creating a variable and
assigning a value to it with the `let` keyword is a statement. In Listing {{#ref fn-main}},
`let y = 6;` is a statement.
```cairo
{{#include ../listings/ch02-common-programming-concepts/no_listing_19_statement/src/lib.cairo}}
```
{{#label fn-main}}
<span class="caption">Listing {{#ref fn-main}}: A `main` function declaration containing one statement</span>
Function definitions are also statements; the entire preceding example is a statement in itself.
Statements do not return values. Therefore, you can’t assign a `let` statement
to another variable, as the following code tries to do; you’ll get an error:
```cairo, noplayground
{{#include ../listings/ch02-common-programming-concepts/no_listing_20_statements_dont_return_values/src/lib.cairo}}
```
When you run this program, the error you’ll get looks like this:
```shell
{{#include ../listings/ch02-common-programming-concepts/no_listing_20_statements_dont_return_values/output.txt}}
```
The `let y = 6` statement does not return a value, so there isn’t anything for
`x` to bind to. This is different from what happens in other languages, such as
C and Ruby, where the assignment returns the value of the assignment. In those
languages, you can write `x = y = 6` and have both `x` and `y` have the value
`6`; that is not the case in Cairo.
Expressions evaluate to a value and make up most of the rest of the code that
you’ll write in Cairo. Consider a math operation, such as `5 + 6`, which is an
expression that evaluates to the value `11`. Expressions can be part of
statements: in Listing {{#ref fn-main}}, the `6` in the statement `let y = 6;` is an
expression that evaluates to the value `6`.
Calling a function is an expression since it always evaluates to a value: the function's
explicit return value, if specified, or the 'unit' type `()` otherwise.
A new scope block created with curly brackets is an expression, for example:
```cairo
{{#include ../listings/ch02-common-programming-concepts/
```

no_listing_21_blocks_are_expressions/src/lib.cairo:all}}
~~~
This expression:
```cairo, noplayground
{{#include ../listings/ch02-common-programming-concepts/
no_listing_21_blocks_are_expressions/src/lib.cairo:block_expr}}
~~~
is a block that, in this case, evaluates to `4`. That value gets bound to `y`
as part of the `let` statement. Note that the `x + 1` line doesn't have a
semicolon at the end, which is unlike most of the lines you've seen so far.
Expressions do not include ending semicolons. If you add a semicolon to the end
of an expression, you turn it into a statement, and it will then not return a
value. Keep this in mind as you explore function return values and expressions
next.
## Functions with Return Values
Functions can return values to the code that calls them. We don't name return
values, but we must declare their type after an arrow (`->`). In Cairo, the
return value of the function is synonymous with the value of the final
expression in the block of the body of a function. You can return early from a
function by using the `return` keyword and specifying a value, but most
functions return the last expression implicitly. Here's an example of a
function that returns a value:
```cairo
{{#include ../listings/ch02-common-programming-concepts/
no_listing_22_function_return_values/src/lib.cairo}}
```
There are no function calls, or even `let` statements in the `five`
function—just the number `5` by itself. That's a perfectly valid function in
Cairo. Note that the function's return type is specified too, as `-> u32`. Try
running this code; the output should look like this:
```shell
{{#include ../listings/ch02-common-programming-concepts/
no_listing_22_function_return_values/output.txt}}
```
The `5` in `five` is the function's return value, which is why the return type
is `u32`. Let's examine this in more detail. There are two important bits:
first, the line `let x = five();` shows that we're using the return value of a
function to initialize a variable. Because the function `five` returns a `5`,
that line is the same as the following:
```cairo, noplayground
let x = 5;
```
Second, the `five` function has no parameters and defines the type of the
return value, but the body of the function is a lonely `5` with no semicolon
because it's an expression whose value we want to return.
Let's look at another example:

```cairo
{{#include ../listings/ch02-common-programming-concepts/
no_listing_23_function_return_values_2/src/lib.cairo}}
```
Running this code will print `x = 6`. But if we place a
semicolon at the end of the line containing `x + 1`, changing it from an
expression to a statement, we'll get an error:
```cairo,does_not_compile
{{#include ../listings/ch02-common-programming-concepts/
no_listing_24_function_return_invalid/src/lib.cairo}}
```
```shell
{{#include ../listings/ch02-common-programming-concepts/
no_listing_24_function_return_invalid/output.txt}}
```
The main error message, `Unexpected return type`, reveals the core issue with this
code. The definition of the function `plus_one` says that it will return an
`u32`, but statements don't evaluate to a value, which is expressed by `()`,
the unit type. Therefore, nothing is returned, which contradicts the function
definition and results in an error.
{{#quiz ../quizzes/ch02-03-functions.toml}}