

Part 07 - Functions

Functions

Definition: Function

A sequence of code which performs a specific task, as part of a larger program, and is grouped as one, or more, statement blocks

Builtin Functions

You have already seen a few functions. `range()`, `print()`, and `join()`.

These are functions built into Boo.

Here's a list of all the builtin functions that Boo offers:

Name	Description	Syntax example
<code>print</code>	Prints an object to Standard Out. The equivalent of <code>System.Console.WriteLine</code>	<code>print("hey")</code>
<code>gets</code>	Returns a string of input that originates from <code>System.Console.ReadLine()</code> - Standard Input	<code>input = gets()</code>
<code>prompt</code>	Prompts the user for information.	<code>input = prompt("How are you? ")</code>
<code>join</code>	Walk through an <code>IEnumerable</code> object and put all of those elements into one string.	<code>join([1, 2, 3, 4, 5]) == "1 2 3 4 5"</code>
<code>map</code>	Returns an <code>IEnumerable</code> object that applies a specific function to each element in another <code>IEnumerable</code> object.	<code>map([1, 2, 3, 4, 5], func)</code>
<code>array</code>	Used to create an empty array or convert <code>IEnumerable</code> and <code>ICollection</code> objects to an array	<code>array(int, [1, 2, 3, 4, 5]) == (1, 2, 3, 4, 5)</code>
<code>matrix</code>	Creates a multidimensional array. See Multidimensional Arrays for more info.	<code>matrix(int, 2, 2)</code>
<code>iterator</code>	Creates an <code>IEnumerable</code> from an object	<code>List(iterator('abcde')) == ['a', 'b', 'c', 'd', 'e']</code>

shellp	Start a Process. Returns a Process object	process = shellp("MyProgram.exe", "")
shell	Invoke an application. Returns a string containing the program's output to Standard Out	input = shell("echo hi there", "")
shellm	Execute the specified managed application in a new AppDomain. Returns a string containing the program's output to Standard Out	input = shellm("MyProgram.exe", (,))
enumerate	Creates an IEnumerator from another, but gives it a pairing of (<i>index, value</i>)	List(enumerate(range(5, 8))) == [(0, 5), (1, 6), (2, 7)]
range	Returns an IEnumerable containing a list of ints	List(range(5)) == [0, 1, 2, 3, 4]
reversed	Returns an IEnumerable with its members in reverse order	List(reversed(range(5))) == [4, 3, 2, 1, 0]
zip	Returns an IEnumerable that is a "mesh" of two or more IEnumerableables.	array(zip([1, 2, 3], [4, 5, 6])) == [(1, 4), (2, 5), (3, 6)]
cat	Concatenate two or more Enumerators head-to-tail	List(cat(range(3), range(3, 6))) == [0, 1, 2, 3, 4, 5]

These are all very handy to know. Not required, but it makes programming all that much easier.

Defining Your Own Functions

It's very simple to define your own functions as well.

declaring a function

```
def Hello():
    return "Hello, World!"

print Hello()
```

Output

Hello, World!

Now it's ok if you don't understand any of that, I'll go through it step-by-step.

1. `def Hello():`
 - `def` declares that you are starting to declare a function. `def` is short for "define".
 - `Hello` is the name of the function. You could call it almost anything you wanted, as long as it doesn't have any spaces and doesn't start with a number.
 - `()` this means what kind of arguments the function will take. Since we don't accept any arguments, it is left blank.
1. a. `return "Hello, World!"`
 - `return` is a keyword that lets the function know what to emit to its invoker.
 - `"Hello, World!"` is the string that the `return` statement will send.
2. `print Hello()`
 - `print` is the happy little `print` macro that we covered before.
 - `Hello()` calls the `Hello` function with no `()` arguments.

Like variables, function types are inferred.

```
def Hello():  
    return "Hello, World!"
```

will always return a string, so Boo will infer that `string` is its return type. You could have done this to achieve the same result:

```
def Hello() as string:  
    return "Hello, World!"
```

Recommendation

If it is not obvious, specify the return type for a function.

If Boo cannot infer a return type, it will assume `object`. If there is no return type then the return type is called 'void', which basically means nothing. To have no return type you can leave off the `return`, or have a `return` with no expression. If there are multiple `return}}`s with different `{{return types`, it will return the closest common ancestor, often times `object` but not always.

Arguments

i Definition: Argument

A way of allowing the same sequence of commands to operate on different data without re-specifying the instructions.

Arguments are very handy, as they can allow a function to do different things based on the input.

arguments example

```
def Hello(name as string):  
    return "Hello, ${name}!"  
  
print Hello("Monkey")
```

Output

Hello, Monkey!

Here it is again, step-by-step.

1. `def Hello(name as string):`
 - `def` declares that you are starting to declare a function.
 - `Hello` is the name of the function. You could call it almost anything you wanted, as long as it doesn't have any spaces and doesn't start with a number.
 - `(name as string)` this means what kind of arguments the function will take. This function will take one argument: `name`. When you call the function, the `name` must be a string, otherwise you will get a compiler error - "The best overload for the method Hello is not compatible with the argument list '(The,Types, of, The, Parameters, Entered)'."
1. a. `return "Hello, ${name}!"`
 - `return` is a keyword that exits the function, and optionally return a value to the caller.
 - `"Hello, ${name}!"` uses [String Interpolation](#) to place the value of `name` directly into the string.
2. `print Hello("Monkey")`
 - `print` is the happy little `print` macro that we covered [before](#).
 - `Hello("Monkey")` calls the `Hello` function with the `("Monkey")` argument.

Function Overloading

i Definition: Overloading

Giving multiple meanings to the same name, but making them distinguishable by context. For example, two procedures with the same name are overloading that name as long as the compiler can determine which one you mean from contextual information such as the type and number of parameters that you supply when you call it.

Function overloading takes place when a function is declared multiple times with different arguments.

overloading example

```
def Hello():  
    return "Hello, World!"  
  
def Hello(name as string):  
    return "Hello, ${name}!"  
  
def Hello(num as int):  
    return "Hello, Number ${num}!"  
  
def Hello(name as string, other as string):  
    return "Hello, ${name} and ${other}!"  
  
print Hello()  
print Hello("Monkey")  
print Hello(2)  
print Hello("Cat", "Dog")
```

Output

```
Hello, World!  
Hello, Monkey!  
Hello, Number 2!  
Hello, Cat and Dog!
```

Variable Arguments

There is a way to pass an arbitrary number of arguments.

variable arguments example

```
def Test(*args as (object)):  
    return args.Length  
  
print Test("hey", "there")  
print Test(1, 2, 3, 4, 5)  
print Test("test")  
  
a = (5, 8, 1)  
print Test(*a)
```

Output

```
2  
5  
1  
3
```

The star * lets it known that everything past that is arbitrary.

It is also used to explode parameters, as in `print Test(*a)` causes 3 arguments to be passed.

You can have required parameters before the *args, just like in any other function, but not after, as after all the required parameters are supplied the rest are past into your argument array.

Exercises

1. Write a function that prints something nice if it is fed an even number and prints something mean if it is fed an odd number.

Go on to [Part 08 - Classes](#)