Lesson 4, Week 2: Functions III (generic functions)

AIM

— Illustrate and explain Julia's use of generic functions

After this lesson, you will be able to

* Use the `::` operator to specify acceptable input types for the values of variables in a function call
* Describe what a generic function is and exactly what constitutes the code of a method of a generic function
* Explain what is meant by the type signature of a function method
* Use `methods` to determine the type signatures of all the methods of a function

The concept of multiple dispatch is much easier to understand from a few examples than from a first-principles discussion.

Example: specifying the value type for input to a function

To double a number means to multiply by two, but to double a word means to make a word twice as long[1]. Let's create a function called `double` with two methods, one for each type of input. We will use the types `Int64` and `String`, and the operator `::` to specify which type we mean. Inlining the functions is best here:

```
double(x::Int64) = 2x
double(x::String) = x^2
```

DEMO: show that `double(7)` and `double("7")` work, but `double(7.0)` throws an error.

By putting `x::Int64` in the argument list of `double`, we specify that this function body should be used for that type of variable. In other words, we specify that if the argument passed to `double` is a value of type `Int64`, then multiplying by 2 is the method to use.

[1] As in the South Africanisms "now-now" and "what-what".

What `methods` tell you about a function

The function `methods` is what you use to find out how many methods a function has and how the argument list of each method is specified.

DEMO: interrogate `include`, `join`

Explain why `join(777, 'A', "wins")` doesn't throw an error.

What is the type signature of a function method?

As you know, the argument list in function call specifies a number of values[2]. The *type signature* of the call is the list of types of these values, in order. For example, if we define a function with `eg3(x,y) = x*y` then the calls `eg3(1, 2.0)` and `eg3(1.0, 2)` have different signatures. But we define a function before we call it. So what is the type signature of a function at the time we define it—that is, what is the type signature of a function method, before it is called?

Let us use the `methods` to interrogate `spin` and `double`, which tell us what the type signatures are that were defined for each of their methods.

[DEMO in the REPL]

We see that `double` is very clear: there is a method for a value of type `Int64`, a method for a value of type `String`. For `spin` the anwer is not as definite: it is true that the signatures cannot be confused, because they use different number input values, but it says nothing about their types. What does this mean?

Well, it means they show differences in their error behaviour: [DEMO: `double('x')` and `spin('x')`]

In the case of the call `double('x')` Julia never even starts executing the function, because none of its methods fit. In the case of `spin('x')` Julia passes the value `'x'` to the only method that uses a single value in its type signature. However, the code in that method throws an error when it gets a character instead of a string.

Review and summary

* The type operator :: inside a function argument list is used to specify the type of an input value
* The method of a function is the code in its code body
* A generic function is a function that can have more than one method
* The function `methods` lists all the valid type signatures of a generic function; each signature corresponds to exactly one method.

[2] Yes, some of the items may be variable names, but it is always the actual value of the variable that is used in the call.