Lesson 3, Week 2: Functions II (user-defined functions)

AIM

— To teach the basics of user-defined functions in Julia

After this lesson, you will be able to

* Explain how to use the reserved keywords `function` and `end` to create a Julia function
* Specify what names are acceptable for user-defined function, and how to specify the argument list in a user-defined function
* Explain how to use the optional keyword `return` to specify the value a user-defined function returns
* Explain how to write an inline function in Julia, omitting all reserved keywords

Functions are hugely important in Julia: they are the main way to organise your code into short bits, they are the main way Julia achieves its spectacular speed of computation and they are also the main reason why large computational projects can be very successful in Julia[1].

The basic `function() ... end` syntax

We start with an example of making a function out of code you've seen before, for reversing a string:

```
str = "abcd"
str[end:-1:1]
```

Recall that the range `end:-1:1` takes steps of length -1 to go from the index `end` to the index `1`.

There are several ways to make functions. The most flexible way uses the reserved keyword `function`.

```
function spin(str)
    y = str[end:-1:1]
    return y
end
```

1. After `function` must come name followed by round brackets (no whitespace), inside the brackets is the *argument list*.
2. The `end` keyword is essential to close the code block opened by `function`

[1] This course is not about Julia's strengths, so we don't discuss these topics any further.

Here we use `return` keyword to return the value of the variable(s) *after* it is what the function returns (note that if you use more than one variable name, the rule is to use commas to separate them). We deal with omitting it later.

Actually, what `spin` does is available from the built-in function[2] `reverse`, which is faster[3]. But the beauty of Julia is that functions are generic: we can easily extend them with new methods. Let's make a function that reverses only a middle bit of the string.

```
function spin(str,k)
    init = str[1:k]
    finish = str[end-k+1:end]
    mid = str[k+1:end-k]
    y = join([init, spin(mid), finish])
    return y
end
```

DEMO:

Note the message after the function is created : `spin (generic function with 2 methods)`.

This illustrates that in Julia, one function name can be used to name functions that do different things. We say that user-defined functions in Julia are *generic*[4] because means they can always be extended with one more method. Julia determines which method to use for a given call from the argument list in the call.

Acceptable function names

Any string that is a valid variable name is also a valid function name, except reserved keywords, type names and a few others special words in Julia that are not important to know[5].

On this course, we discourage variable names that use exclamation marks and underscores. Although they are valid, Julian style is to use them in special ways that aren't part of this course.

Using a `.jl` file for one or more functions

Of course, if you want to retain the code of a function, you should put it into a `.jl` file, and use `include` to make the function available at the REPL. If you want to put several functions in one `.jl` file, no problem, just do it.

DEMO: both functions in a single file; notice how the name of the file that contains the functions doesn't have to use the function names at all.
Using `include` to make the functions available.

[2]There are very many built-in functions in basic Julia, and many more in the packages. If you continue to code in Julia, you should certainly study and use them.

[3]Speed however is not our aim in this course.

[4]So are almost all Julia functions, and the exceptions are highly technical and far beyond this course.

[5]If you try to use one of them, Julia will give you a clear error message.

Working with functions: testing, debugging

Testing is essential when coding[6].

We should at least test that `spin` does do the reversals we expect—both methods!
We should also check that they fail in the ways we expect.

DEMO in the REPL

Debugging now involves more than just fixing invalid code: we must also fix valid code that does something other than what we want.

Let us change the `.jl` file so that the code is valid but the output is wrong.

DEMO involving both the REPL and the text editor

Omitting the `return` keyword; inline functions

If there is no `return` keyword in your function body, it will still return a value, namely the very last one that was calculated.

[DEMO: modify one or both `spin` functions this way]

This paves the way for *inline functions*: you can omit all the keywords, and just use round brackets after the name, an equals sign, and an expression.

[DEMO: `spin(str)` as an inline function]

This is often very convenient convenient in writing code: instead of one long complicated function, use inline functions for code that occurs often in your function.

There is one more way to write functions: anonymous functions, where you don't even have to use a name. However, this is is mainly for iteration, so we prefer to discuss them in the lecture on iterators[7].

[6] The Julia community has an extremely strong culture of testing and has lots of ways to support and encourage testing.

[7] Of which you've already seen one, namely comprehension.

Review and summary

* You can write a function using the `function() ... end` syntax
* The function body consists of the lines *after* the input argument list and *before* the closing `end`
* The `return` keyword specifies what the function returns; if it is omitted, the function returns the value returned by its final line
* Functions are generic: they can have more than one method; the pattern of input values decides which one is used
* Testing is essential for newly created functions
* A `.jl` file can contain many functions
* With inline functions, you omit the `function` and `end` keywords; it is intended for functions with one-line code bodies

It is essential that you become very comfortable with writing function code in the two ways discussed in this lesson. Please make sure you do the exercises and the self-assessed assignment.