Lesson 2, Week 1: Deconstructing Lesson 1

AIMS:

— to deconstruct the code in Lesson 1
— to give details on expressions in Julia
— to convey the importance of valid code
— to start the process of carefully learning the Julia language

After this lesson, you will be able to

* give a few examples of valid expressions in Julia
* explain the structure of a variable in terms of its name and its value
* explain exactly what a string value is in Julia
* explain in broad terms what a function is, and how Julia recognises a function
* explain how operators differ from other kinds of function
* give examples of some Julia delimiters and of their use

Review: the code of Lesson 1

In Lesson 1, you ran the following code at the REPL

```
mystringexample1 = "Hello, world"
```
... line 1

```
println(mystringexample1)
```
... line 2

You also created a file which we will refer to as `myfile.jl`, although you may have used a different name[1].

```
include("myfile.jl")
```
... line 3

[1] In this lesson, whenever we mention `myfile.jl`, please realise that we are referring to the file you created, and in your mind (and in the code examples), replace it with the name of your own file.

Deconstructing line 1, which created a variable

How to read line 1: it has a left hand side, a right hand side, and the equals sign connects them.

The right hand side is a string value (more on this below): `"Hello, world"`.

The left hand side is the name of a variable: `mystringexample1`.

The `=` sign binds the variable name to the given value.

This actually changes some of the memory in your computer. Technically, the equals sign is a special kind of function, namely an operator, and its full name is "assignment operator"[2].

We say that line 1 *created* the variable because before line 1 `mystringexample1` was not part of the namespace. We could assign a new value, for example with the line

```
mystringexample1 = "a new value"
```

and in this case we would not be creating the variable, merely binding the existing name to a new value. This new value need not be string, by the way, it could for example be a number

```
mystringexample1 = 1.1111
```

or indeed any other kind of value that Julia can work with. As noted in lesson 1, names cannot be removed from a namespace, only added. In this course, we work only with the top-level namespace, which lasts as long as your Julia session.

Deconstructing the string value

A string is a sequence of characters. In Julia, as you've seen, we indicate a string value by enclosing it in a pair of double quotes[3].

We have briefly discussed characters before. Note that in Julia, a character is always enclosed in single quotes.

[DEMO: `a = 'a', b = "a"`]

In this course, we will use only the characters available with one keystroke[4] of the international keyboard to form strings. But many more characters are valid in Julia[5] and later on we will briefly

[2] All computer languages have at least one assignment operator. They differ in the fine details of how they affect the memory inside your computer, but in this course, we stay away from these details. All you need to know is that in Julia, the assignment operator binds the value on its right to the variable name on its left.

[3] Julia has very many kinds of values; in this course we discuss only a few of them. See the discussion of types in Lesson ...

[4] Perhaps modified via Ctrl or Tab key.

[5] Including alphabets like Greek, Arabic, Sanskrit and many others. Indeed not only alphabetic characters but also non-alphabetic characters like those used to write Mandarin are valid characters in Julia.

show you how to make them. You may like to start using them in your variable and function names, but learning to do so is your own project, it is not part of this course.

Deconstructing the variable name

Only some of the characters that Julia accepts in string values are allowed in variable names.

Variable names have to start with a letter[6] and must continue with letters, digits or underscores or the exclamation mark. In this course, we use only letters from the Roman alphabet, but Julia accepts more letters than those.

Best practice is to use only lower case letters and digits, and use descriptive names such as `mystringexample1`[7].

Line 1 is a valid Julia expression

This is extremely important: when Julia processes valid code, the computer changes—and some of the changes achieve the purpose of the code (printouts, calculations, pictures . . .). Line 1 is valid because:

- Line 1 consists of symbols that Julia recognises.
- The symbols are combined into three valid groups.
- These three valid groups are combined into a single valid expression.

Question: which symbols does Julia recognise?
Answer: very many! But in this course we will only use symbols that are visible on a standard international keyboard, all of which Julia recognises.

Question: which are the valid groups of symbols in line 1?
Answer: three of them, namely `mystringexample1`, `=` and `"Hello, world"`. That is, Julia recognises a valid variable name, a valid operator, and a valid string[8].

Question: why is this combination of valid groups of symbols a valid expression?
Answer: because the `=` operator can work that way. Actually, it can *only* work that way: a name on the left, `=` in the middle and a value on the right.

It is only when all the parts of the expression are correctly recognised by Julia, and combined according Julia's rules, that we have an expression which is valid Julia code. The rules for making valid code are extremely strict, we discuss why that is so in Lesson 3.

[6] And some other characters, but in particular not a digit, see the Julia documentation for details.

[7] The Julia community generally observes this rule. This allows the exclamation mark to have a very special meaning which we will see later. Underscores are strongly discouraged in user code, although the rules allow them; the idea is to limit underscores to special meanings in the internals of Julia.

[8] The spaces between are not essential, as would Julia recognise from other clues. However, presence/absence of space does sometimes matter in Julia, as we'll see later.

Finally, let's note that some parts of line 1 would by themselves be valid code, namely the name and the value. A valid expression can be part some larger valid expression.

Evaluating invalid code generates error messages

Here is a bit of jargon: we say that Julia evaluates each expression it gets. This simply means that Julia tries to take the actions that the code instructs it to do.

Invalid code such as the following lines

```
=
= "Hello, world"
mystringexample1 =
mystringexample1 "Hello, world"
"Hello, world" = mystringexample1
```

generate error messages and nothing else[9] from Julia. In lesson 3, we start you off on reading error messages and debugging invalid code.

A subtlety explained

However, and this may surprise you,

```
Hello, = mystringexample1
```

is valid code. This is not because `Hello,` is a valid variable name, but because of a special role for the comma. Here, because it follows a valid name on the left of the assignment operator, the comma indicates that Julia should perform multiple assignment. Let's use an example with two names on the left:

```
Hello, world = mystringexample1
```

... line 4

Note that line 4 introduces a new form of assignment: on the right is not a value, but instead a variable name. No problem, Julia just uses the value that is bound to the variable name.

DEMO: we evaluating line 4, interrogating the variable names and values it creates

Multiple assignment works like this: on the left hand side of `=` you have variables separated by commas[10]. Once it has the list, Julia looks for values on the right hand side. It may seem odd that a single string value can supply several separate values, but remember that a string is a sequence of characters. Since the left hand side is a sequence, Julia treats the right hand side as a sequence. As you can see, superfluous items on the right are ignored.

Using commas in this way to do multiple assignment is one of the ways in which Julia allows you to

[9] This cannot be quite guaranteed, but is definitely what the makers of Julia want!

[10] As you have seen, you could have just one variable followed by a comma, and then the = sign.

create code that is very compact, and also often quite easy to read. If done well, it can really help you to write code that other people like to read, which can greatly ease collaboration. This includes collaboration with yourself, some months or years later, when you try to adapt your old code to a new purpose.

> Don't worry about the difficulty of writing a valid expression in Julia. As you saw in Lesson 1, it can be quite easy. Julia, like any computer language, allows extremely long valid expressions. And yes, to ensure that such an expression is valid can be very hard. But all that is beside the point. You can do an enormous amount with short, simple expressions!
> And you don't have to learn all of Julia at once. In this course, we will introduce you gradually to more and more ways of forming valid expressions, and never too much at any one time.

Line 2 calls a function

When Julia code tells a function to do something, we say that it calls the function. Here we are calling the function `println` with some input. This is input is `mystringexample1`, i.e. the name of a variable.

For this to be valid code, `println` must be able to format the value of that variable[11]. Calling a function in Julia makes things happen.

You should think about this for a moment: the function call `println(mystringexample1)` does several things: first it accepts the variable name, then it fetches the value of the variable, then it formats it—in this case there is very little to do—then it prints the formatted string on the screen, followed by an empty line before the next `julia>` prompt.

How does Julia know that a bit of code is referring to a function? Simple: the code is a valid variable name immediately followed by parentheses — that is, without any space after the function name the next character is `(`. After that comes the input to the function, and after the input the closing `)`.

However, `println` doesn't actually need any input: [DEMO: `println()`, `include()`]. You see that `println()` behaves as if you had given it an empty string: it prints a line with nothing in it, then it skips to a new line, then it skips to the `julia>` prompt. On the other hand, `include()` throws an error.

There is one exception to the rule that Julia recognises a function via the `(` that follows a name. Operators, as we noted, are a special kind of function. They are (mostly) mathematical symbols such as `-`, `+`, `>` and the expressions they form have rules that follow the usual mathematical meanings fairly closely[12]. In this course, you will learn many more built-in Julia functions, and you will also write your own. A very large part of Julia code is written by means of functions.

[11] Plain strings such as `"Hello, world"` here are the easiest of all values to format.

[12] But be careful: the mathematical meanings are just an indication. The rules have to be followed exactly, so you have to know them!

Final deconstruction: delimiters

The parentheses we use in `println()` play an important role in Julia: they are delimiters. They serve to tell Julia where input to the function starts and ends. Similarly, the double quotes around strings are delimiters, and so is the comma when it is used to do multiple assignment.

Please take careful note of what you typed to make valid code: using valid characters, you typed values, names, operators and delimiters. That covers all of the valid code we use on this course[13].

Review and summary

- Julia code consists of valid expressions.
- In Week 1, we use valid expressions that contain values, names, operators and delimiters.
- Names in Julia have to start with a letter, and continue with letters or digits or underscore or exclamation mark.
- A variable is a name bound to a value.
- Calling a function means following its name with parentheses around the values and/or variable names that you pass to the function.
- Operators are a special kind of function that do not need parentheses—usually they are symbols like `=`, `+`, `<` etc.
- Delimiters like a `" "` pair around a string value and parentheses `( )` around function input help to structure to Julia code.

[13] As noted, we use only a small subset of the valid characters in Julia. Similarly, we use only some of Julia's operators and delimiters.