Lesson 1, Week 1: A minimal working example

AIM

* To provide an example in order to start for talking about Julia

---

In this our first lesson, we draw extensively on Lesson 0—the one on what you have to have in place to be ready for this course[1].

I also want to remind you of the course slogan: **small steps, no gaps, make sense always**. The aim is to keep nervous beginners on board!

In this lesson we start from zero, so we can't keep to all parts of the slogan fully. The idea is first to see coding in action in this lesson, and then explain it in detail in lesson 2.

Entering REPL code

Open the REPL, enter `"Hello, world"`.

DEMO: In the video of this lesson, we show you exactly how to do this.

`"Hello, world"` is a string value. Julia has many other kinds of values: we'll see some of them, such as number values and character values[2].

Now enter `mystringexample1 = "Hello, world"`. This is called *assigning a value to a variable.*

IMPORTANT: The `=` sign binds the string value on the right hand side to the variable name on the left. This changed your computer's memory in three places:

- The name `mystringexample1` was put into what is called the namespace[3] .
- The string value `"Hello, world"` was created[4].

[1] Firing up Julia and getting an REPL; editing plain text files and storing them in a special folder for this course.
[2] That is, values that are characters.
[3] Actually, when Julia is running it can have several namespaces, but that is an advanced topic we do not address on this course. Once a name is in a namespace, it will stay there until you shut down the whole namespace. Closing down your Julia session also closes down all the namespaces.
[4] Separately from but in the same way as above.

- The `=` sign between the name on the left and value on the right created a binding between the name and the value.
- By binding the string value to the name, Julia is keeping the string value in your computer's memory so that the value is available in case it is needed later.

Enter `println(mystringexample1)` .

DEMO: `println` is built-in function

What happens when this line is run[5]:
The function `println()` receives the variable name `mystringexample1`, fetches its value (which is a string), reformats it, and on the screen displays the string followed by blank line.

Functions are very, very important in Julia. Many are built-in, such as `println()`, but Julia programs also create many more. You will learn a lot about functions in Julia on this course!

Creating and running a code file

Finally, create `myfirstfile.jl`, as a plain text file (NB!) containing exactly the two lines of code we used above, save it in your course folder[a]. Make sure your course folder is your working directory[b], and enter `include("myfirstfile.jl")`

DEMO: in the video, we show that the result is the same as for the REPL code we used earlier.

[a] That is, create a Julia code file—such files are one of the topics of Lesson 0.
[b] Use `pwd()` to check what your working directory is, and `cd()` to change it.

Congratulations! Your first Julia program! Coding is that simple.

Review and summary

* `"Hello, world"` is a string value.
* `println()` is a function.
* `mystringexample1` is a variable name.
* `=` is assignment: the value of the right hand side binds to the name on the left hand side.
* The function `include()` runs the lines of code it receives from a Julia code file.
* A Julia code file is a plain text file with the extension `.jl`

What we did in this lesson is what we'll do over and over as the course proceeds: some new ideas, some examples (which you to try as the lesson proceeds), and some code files for you to write and execute.

Please do the quiz, the exercise and the self-graded assignment before going on to Lecture 2 of Week 1. They're very short! It really is best to do them before going on.

[5] People also say: "when this line is executed", and "when this line is evaluated".

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.

Lesson 3, Week 1: Error messages and debugging

AIM

— to learn to read error messages

— to learn line-by-line method of debugging

After this lesson, you will be able to

* find useful information in Julia's error messages

* find errors in a single line of Julia code and fix them

* step through Julia code line by line, fixing errors as you find them

* occasionally go back to fix errors you missed

Debugging some almost-Lesson-1 code

Suppose by mistake we tried to run Lesson 1 using the code below[1].

```
mystiring1 = "Hello, world
prinln(mystring1)
```

You find it doesn't run (and, depending how you are trying to run it, you may see an error message).

By examining the code line by line you can find out why. Here, when we try to run the first line from the REPL, it hangs, without any error message. Your job now is to find out why, and you start with a careful look at the code. Is it valid code? No! There is no closing double quote character to indicate the end of the string value. As Julia sees it, you are still entering characters in a string[2].

Fix that line by adding the missing double quote character at the end.

[1] NB: this works best if you start with a fresh REPL. It contains typos, you should be careful to type the example below exactly as it is, typographical errors and all

[2] If you put these two lines in a .jl file and tried to use `include()` run it, you would get the error message " LoadError: syntax: incomplete: invalid string syntax" which is more helpful, but not very much more. Fact is, there are many ways that syntax errors can occur, and Julia does not try very hard to tell you exactly what the error is.

Here's a good tip: also add a semi-colon after the closing double quotes. In the REPL, this suppresses the output of that line, which means you have less to read and see fewer distractions. Of course, in some cases you want to see the output of evaluating a line of code, this isn't always the right thing to do.

Ok, so now the code becomes

```
mystiring1 = "Hello, world";
prinln(mystring1)
```

When we run the first line in the REPL, as expected see no output. So we run the second line. Oh dear. Suddenly lots of lines on the screen and some of them in red!

The line starting with ERROR is the important one[3]: this is the error message. In it, Julia gives you some indication of what the problem might be. Here, the message is `UndefVarError: prinln not defined`. But of course that is easy to fix also, so now the code becomes

```
mystiring1 = "Hello, world";
println(mystring1)
```

And now we get an error: again the message `UndefVarError: mystring1 not defined` . How unexpected! And it seems wrong: the line is valid Jullia code! When we examine the `println()` line there seems to be nothing wrong with the variable name, how can it be undefined?? This often happens in debugging: some errors mask others. You fix the first ones you find, and eventually they reveal errors earlier in your code. Here of course there is a spelling error in the line above it. If we don't spot that right away, the fact that the error message says `mystring1 not defined` should help. It means this is the first time that Julia sees the name `mystring1`. Whatever we believed about creating it earlier is false: either there is no line at all for creating it, or the line is wrong. Find the line where it should have been created, and fix the error. Finally, all correct when we change for the last time to

```
mystring1 = "Hello, world";
println(mystring1)
```

Some points about debugging

Line by line examination in this way is by no means the only way to find bugs, but it is a good place to start learning how to debug code. You should practice it a lot and become skilled at interpreting error messages.

In fact, debugging is an absolutely vital skill for programmers. Just as for writing code, here are many, many styles of debugging[4]. But at the heart of them all is the ability to read through code with a very critical, searching eye. That ability is what you should cultivate.

[3] So important that in most terminals it come up bright red.

[4] As we'll see in the next lesson, writing code is fundamentally an act of creative expression. Debugging too allows you to go about it your own way.

If you go on to program a lot, you will probably want to learn about software to help you with the task. There is quite a lot out there[5], but in this course we do not go into that.

Review and summary

* Debugging line by line is a very useful skill to learn.
* Error messages in Julia contain useful hints, but also a lot of other detail.
* Error messages in Julia[6] cannot guarantee that they point at the actual error.
* Line by line output is not quite the same in the REPL as it is for running a .jl file with the same code.
* For line by line output, it can be helpful to suppress output with a semi-colon at the end of a line.

[5] For Julia, the main supports are the `Debugger.jl` package, which is very useful but takes a while to learn, and the `Juno` development interface, which is not specially for debugging but has a lot of features that can help with it—and these can be extended by using `Debugger.jl`.

[6] Or any other language, for that matter.

Lesson 4, Week 1: Programming is applied formal logic

This is the only theory lesson in the whole course. As such, it is very, very important!

AIM

— to understand the difference between logic and formal logic

— to understand how formal logic makes computer language possible

After this lesson, you will be able to

* explain what we mean by the word logic
* explain the difference between logic and formal logic
* explain why any computer must embody formal logic
* give a small example of a language that applies formal logic
* explain why writing computer programs is expressive, in the same way as writing poems or music

Why this lesson?

Programming often feels very strange to beginners, and if they haven't worked much with mathematical formulae, it can feel utterly alien.

This lesson cannot make the initial experience any less weird. Weird is part of the territory. But by explaining that it isn't arbitrary, nor aimed at keeping people out, but instead an essential aspect of any computer language, I hope to encourage you to persist until the Julia way of writing programs starts to feel natural. At that point, you will have learnt a new languge.

That is entirely serious: Julia, as I hope you will see below, is as rich and expressive in its own right as any natural language, such as Spanish, Mandarin and !Kung. Apart from implementing formal logic, there is no real difference with other languages. You could use it to write advertisements, love letters, novels, whatever! But mostly, since the formal logic can be implemented on a machine, you use it to write programs that make machines do things you want them to do.

Logic: the study of correct forms of reasoning

Reasoning is part of everyday life and language. It is simply the process of trying to make sense of things and expressing that as clearly as possible.

Unfortunately, there are many ways people use to make sense that actually are mistakes in reasoning. For example, it makes sense that raining wets grass, and this may lead people to reason as follows: if the grass is wet, there has been rain. Although this may often be true, it isn't always true: rain is not the only way grass can get wet. Strictly speaking, it is not correct. Reasoning correctly is quite hard!

As far we know, the first people who became aware of this *and* made a study of correct forms of reasoning lived approximately 100 generations ago. This occurred separately, in at least three places: in ancient China, ancient Greece and ancient India rules were formulated for reasoning correctly.

The word logic is used in many different ways in ordinary language. In this course, we use it in only one sense: the study of correct forms of reasoning.

Formal logic: using symbolic formulae to study and apply logic

Formal logic is simple to define: it uses symbols and formulae to study reasoning. This is fairly new discipline, it started only about 6 generations ago[1].

The aim of making logic formal was absolute precision in reasoning and absolute certainty about its correctness. It aims at this by using rules similar to the rules of algebra[2]. By doing so, it reduces logical analysis to calculations which can be independently checked. In other words, it removes the need for intuition and insight from logical analysis.

Formal logic has the strange property that something obvious and something baffling can sit side by side. For an example of the obvious, see the NOT, AND and OR truth tables below. Unfortunately, we you'll have to take our word for for the ease with which baffling things come up quite unexpectedly (or read *Logicomix*, in particular pages 164 to 168, about paradoxes).

Truth tables for `NOT`, `AND`, `OR`

In two-valued logic, which is by far the most common type of formal logic and also what Julia uses, there are two constant values, we denote them with `true` and `false` [3]. A statement is then something that is either `true` or `false`. As noted above, for a formal logic we need a symbol to stand for a statement, for example we can use P to stand for the statement "The earth spins on its axis" and Q for the statement "The earth is round"[4].

[1] See *Logicomix* by Doxiadis and Papadimitrou for an excellent and very human account of some of the most important episodes in the early history of formal logic.

[2] This goal has been considerably refined, because it was shown that a system cannot be consistent as well as complete and applicable to all of mathematics. For computers, it is not necessary to produce *all* correct reasoning—it is enough to ensure that all the reasoning in your computer is completely correct.

[3] In Julia code they are the values `true` and `false`.

[4] You may disagree with either of these; the first is much clearer than the second. Of course the earth is not perfectly round, but for practical purposes the imperfections are negligible, so it is perfectly reasonable to say that Q is true. This

So now we have two symbols P and Q. Here are three more: $\neg$, $\wedge$, $\vee$. With these, lets look at four valid formulae: $\neg P$, $\neg Q$, $P \wedge Q$, $P \vee Q$.

In words, these are "The earth does not spin on its axis", "The earth is not round", "The earth spins on its axis and the earth is round", and "The earth spins on its axis or the earth is round". That is, $\neg$ means `NOT`, $\wedge$ means `AND` and $\vee$ means `OR`.

But what do these mean, in terms of formal logic? Well, they (may) change the truth values. For example, if P is true then $\neg P$ must be false. A good way to summarise exactly what they mean is by means of truth tables. In a truth table, we specify the truth value of all valid combinations.

So the truth table for $\neg$ (equivalently, `NOT`) is as follows (note that *two* lines are needed to get both possibilities):

P	$\neg P$
`true`	`false`
`false`	`true`

The truth table for $\wedge$ (equivalently, `AND`) requires four lines:

P	Q	$P \wedge Q$
`true`	`true`	`true`
`true`	`false`	`false`
`false`	`true`	`false`
`false`	`false`	`false`

In other words, the combined forumula $P \wedge Q$ is `true` only when both are `true`, and `false` otherwise.

The truth table for $\vee$ (equivalently, `OR`) is:

P	Q	$P \vee Q$
`true`	`true`	`true`
`true`	`false`	`true`
`false`	`true`	`true`
`false`	`false`	`false`

In other words, the combined forumula $P \vee Q$ is `false` only when both are `false`, and `true` otherwise.

Because they may change truth values, `NOT`, `AND` and `OR` are called operators. To be more specific, they are logical operators.

The symbols we've seen are standard for research papers in formal logic, but not in progamming. In Julia, the symbol for `NOT` is `!`, `AND` is `&&` and `OR` is `||` . We'll see these again later; the fact that we need two characters make one symbol is of no significance.

Why formal logic needs a formal language

For formal logic to work, every expression must consist of symbols. In fact, it must consist of symbols that formal logic recognises—in exactly the same way that every expression in Julia must consist of symbols that Julia recognises[5].

sort of thing is why ordinary language and everyday reasoning are very hard to represent in a formal language.

[5]Introductions to formal logic have to solve a bootstrap problem: until you know the symbols, you can't do formal logic. They tend to start as we did here: by translating some very simple sentences into symbols. This has the effect of making the very simple seem complicated. Weird, but very hard to avoid.

A formal language makes it possible to say exactly what symbols and expressions are valid in that language. It has to go beyond the three operators we've seen above. That is, in order to express something, one has to use symbols that can carry truth values. Above, we used P and Q for that, but that was for convenience only. There are much better ways, and Julia is one of those.

Let's repeat that: Julia is a formal language[6]. It uses characters to form values and names, and then it uses delimiters and operators to form the names and values into expressions. The rules of the formal logic that governs Julia then determine whether the expression is valid or not.

In other words, you can write an unbelievably large number of Julia programs, but in order for them to consist of valid expressions, you have to follow the rules of Julia's formal language and formal logic.

Formal languages can generative and make computers possible

By "generative" I mean that one can make new expressions from old ones. A useful language with only a few valid expressions can certainly exist, but Julia and all major computer languages allow a great many valid expressions, and also set no limit on how many such expressions are used to write a program.

Writing a poem is also the result of a generative process: you add one word after another, one line after another, and there is no set limit on how long your poem may be.

But a formal language is not just generative: because the rules are completely fixed and explicit, they can be mechanised. They can be made to be part of a machine[7]; in such a machine you have a way to represent the values `true` and `false`. Then when you apply `not` to a `true` you get the value `false`, and so on. Such a machine can produce and evaluate all the expressions of a formal language and its the formal logic.

At the level of a microchip, all modern electronic digital computers contain many millions of tiny electrical circuits that are nothing but electronic versions of `NOT` , `AND`, `OR`[8]. Formal logic is what makes a computer possible; all modern computers are applications of formal logic.

A formal language with just a few letters and rules

Let us look at brief example of a formal language, let's call it AdHoc1. It will be very simple, just a few letters and rules. They allow the formation of some sentences in English, but also of expressions that are very far from being English. That is, the valid expressions in this language include some English but not all of it, and includes non-English also. To specify AdHoc1, we give a full list of symbols and a full list of how rules for building expressions.

The symbols are a few characters: the letters `a c e i f h l n s t` and the space character.

The rules are

[6] In fact, every computer language is a formal language.

[7] The earliest programmable computer design was by Charles Babbage over 150 years ago, for a machine to be driven by steam.

[8] Of course, they contain much more than just these operators, but this is not a course in computer design.

1. Some letters are also vowels, they are: `a e i`.
2. Some letters are also consonants, they are `c f h l n s t`.
3. An expression must start with a consonant.
4. An expression may not contain two vowels next to each other.
5. Any expression can have a character added to it, except that
   (a) No expression can have more than 100 characters
   (b) No expression can end with the space character

Note that some expressions made with AdHoc1 look like sentences in English, for example "that is it" and "that is it i think". But strangely enough, "i think that is it" is *not* a valid expression in AdHoc1, because it violates rule 3. Some amusing expressions are possible, such as "the think that i think that i think that i that think thinks". Also the famous paradox "this sentence is false" is valid AdHoc1.

However, by reading that last sentence as a paradox we are reading it as English. This is not part of AdHoc1, and requires an extension that applies AdHoc1 expressions to English. But most AdHoc1 expressions are not English, for example "ticillllll eee", so this extension is very informal[9] and relies a good deal on us as people who can read English.

There are some other patterns to note (or oddities, if you prefer): you cannot form the empty expression (because of rule 3), you can't write more than one line (there is no newline character), there are no delimiters so you cannot do punctuation. All one has so far with with AdHoc1 is a a set of rules for making

Let us also note that AdHoc1 cannot be a computer language, because it has no way of making the computer do anything. For that, there needs to be a number of expressions in the language that cause something to happen inside the computer. An example in Julia is the `println` function. In fact both lines of code in your first computer program makes something happen in or via the computer. Most of the details of this is hidden from the user of Julia—and this is one of the reasons for choosing Julia as a first programming language to learn.

Programming, poetry and truth

> 'Beauty is truth, truth beauty,—that is all
> Ye know on earth, and all ye need to know.' (Keats)

As we have repeatedly seen, a programming language like Julia allows you make many, many expressions. When you choose what names your variables are to have, which functions to employ, how to structure the logical operators in your program, and so on, you are using the language to express yourself. You are doing much the same as a poet.

Moreover, you cause things to happen: certain numbers are calculated, pictures are drawn, documents are printed. A poet aims at different effects: reactions from readers/listeners, memorable descriptions, catching the mood. But like a poet, the programmer is using their skill to achieve definite purposes.

[9] Actually, no formal language exists that produces all the valid sentences in English and only those sentences. This is partly because people cannot agree on what all the valid sentences in English are.

Let us not forget truth: it is central to logic, as we saw above. Computer programs utterly rely on the programmer keeping a very strict eye on whether their logic maintains truth. Paraphrasing Keats, one might say: "Code is truth, truth code,—that is all you need at the keyboard".

The kind of truth that is embodied in fine pottery is not same as the truth in beautiful poetry, and the truth that is captured in a computer program is far from both of these. But in the end, all three these forms use the expressive power of the medium to get at truth.

Programmers are, in that sense, like artists and poets[10].

Review and summary

* Logic (in this course) is the study of correct forms of reasoning
* Formal logic is the use of formulae to do logic
* Truth tables spell out exactly how logical operators work
* To be useful, a formal logic must be part of a formal language
* A formal language is a set of symbols and a set of rules for combining them
* Julia is itself a formal language
* A programming language is capable of endless variety of expression
* Writing code is a creative as much as writing poetry or music.

[10] But also like storytellers, stand-up comedians, composers and basket-weavers. All of these aim at certain effects which in some sense set a standard of truth.

Lesson 5, Week 1: Strings I (literal basics)

AIM

— to understand what a literal string is

— to learn to combine the string operators `*` and `^`

— to learn how to enter Unicode characters at the REPL

— to learn how to use `+` and `-` as operators on characters and integers

After this lesson, you will be able to

* build literal strings using characters, other strings, and operators
* insert some characters by another means than finding them on your keyboard
* do some very simple character arithmetic

String literals

Consider assignment of the string value `"abcdef"` to the variable `var1`, which as you know, is done with the code
`var1 = "abcdef"`

As you also know, this means that the string value `"abcdef"` is physically stored somewhere on your computer, where it takes up a small amount of memory. To be precise, the memory of your computer is modified so that in some part of it, the character `'a'` is followed by the character `'b'` and so on[1].

A string value stored in memory this way is a string literal.

The string operators `*` and `^`

When used as a string operator, `*` combines two strings. [DEMO: some more examples]
For example, `"Julia " * "programming"`

[1] Of course, what is actually stored is a sequence of 0s and 1s that Julia knows should be interpreted in a particular way.

The technical term is *concatenate*. Thus the in the code `"Hello" * ", " * "world"`, three strings are concatenated into one.

Note that for the purpose of concatenating strings, characters can also be used.

DEMO

What happens with code like `"a" * 'b'` is that first the character `'b'` is converted to the string `"b"` and then the two strings are joined.

[DEMO: more examples]

The string operator `^` is a bit different: its input is one string and one integer. Consider the code `"my" * "my" * "my"`
Here, a string literal is repeated three times. Instead of typing it out three times, one can just use `"my"^3`.

Note that these operators can also be used with variables whose values are string literals:

```
x, y, z = "string 1 ", "this is a string ", "but this?  what's this?"
```

```
x * y * z
```

```
(y*z)^3
```

Characters not directly from the keyboard

For example: you can get Greek letters from an standard keyboard that shows only Roman letters. The letter α is available as follows: at the REPL, type `\alpha` and then press Tab.

In fact, you can type only `\alp` and press Tab. The REPL then shows you all the options that start like that. Pick the one you want, type in the next letter, and try again. Continue until it completes correctly, then press Tab again.

Which characters are available by this method? There is no easy answer. If you know the LaTeXtypesetting system, you can try any character code from there, it should be available. You can also try to explore the system by typing just one or two letters after the backslash, and look at all the options[2].

Finally, for those of you who want more, some of Unicode is available. If you know a UTF-8 code of a character, you can try it. You still use a pair of single quotes, and inside it you put `\u...` where the dots stand for an integer with one to four hexadecimal digits. We won't use any of these characters on this course.

WARNING: some of these characters may be available at the REPL, but not in your text editor. In that case, you won't be able to use them in a `.jl` file. This is one of the differences between the REPL and code files to watch out for.

[2] Gets tedious quite quickly!

Some very simple character arithmetic

For completeness[3], we mention that `+` and `-` can act like operators on a character, for example `'a' + 1`.

As you see, this shifts the character to a nearby character—the distance is determined by the integer you add or subtract. Note that the order doesn't matter[4]: `'Z' + 22` gives the same result as `22 + 'Z'`.

Review and summary

* A string literal is simply all the actual characters of the string, all together in one place in the computer's memory
* The operator `*` concatenates strings
* The operator `^` makes a new string by repeating one string (inputs: string, number of repeats)
* Characters not on an international keyboard[5] are available via `\`, followed by some letters, followed by Tab.

[3] That is, we don't use this feature on this course, but it feels like a gap to leave it out.
[4] As opposed to `*` when it is used as a string operator.
[5] None of these are used on this course, except as examples.

Lesson 6, Week 1: Strings II (escape sequences)

AIM

— to understand what an escape sequence is

— to learn the escape sequence `\"`

— to explain the difference between a string with an escape sequence and its formatted appearance

— to learn the escape sequence `\n`

— to learn the escape sequence `\t`

— to learn the escape sequence `\\`

After this lesson, you will be able to

* Say why `'$'` and `\` behave oddly when used to make a string value in Julia
* Define an escape sequence in Julia
* Understand that formatting a string with an escape sequence gets rid of the escape sequence
* Effectively use the escape sequences `\"`, `\n` and `\t`
* Understand that `\` can be used to escape itself

Neither `$` nor `\` come up literally if they're in a string

DEMO: here are two invalid strings: `"\"` and `"$"`

This lesson is about strings like these, where what you see is different from what you type. In fact, it will take us two lessons to cover this topic. In this, we look at what is called *escape sequences*: they all start with a backslash. In the next lesson , we look at what is called *string interpolation*, which involves the dollar character.

What you'll see is that the formatted form of a string is affected by the escape sequences in the string. You already know that the formatted form is different from the string literal itself, because you've seen the difference between `"Hello, world"` and `print("Hello, world")`.

DEMO: a few more examples of the literal vs formatted version of a string

Making and printing a string with the escape sequence

In Julia, all escape sequences start with `\`. A useful and very simple escape sequence is `\"`, which allows you to have double quotes *inside* the formatted version of a string.

DEMO: you cannot get the formatted string `He said "Hello, world"`
with the code `x="He said "Hello, world""; print(x)`.

The reason the code above fails is clear: the first double quote character starts a string, and then the second one ends it. It is worth tarrying a while over the error message syntax: cannot juxtapose string literal. The string has ended with that second double quote, so what follows is new code. Julia could wait to see what the new code might be, but since this is such a common source of bugs, there is a complicated rule for detecting whether this is really the end of a string. Here we have a letter immediately after the double quote, hence the message. A space or a delimiter gives a different error message, and so on.

DEMO: the code you need is `x="He said \"Hello, world\""; print(x)`

We need to read this with some care: the two double quotes we want to see in the formatted version are said to be *escaped* by the backslashes preceding them. The combination `\"` is an escape sequence.

The escape sequences `\n` and `\t`

The escape sequence `\n` shifts the bit of string that follows onto a new line[1].

DEMO: `x="Hello, \nworld"; print(x)`

[1] And therefore is oftened called the newline character—yes, the two characters together are thought of as one character.

With this in hand, we can easily write multiple lines in a single string[2], for example

```
doggerel = "Errors are red, \nsome things are blue, \nI love coding \nand so should you"
println(doggerel)
```

The escape sequence `\t` inserts empty space up to the next tab stop:

```
tabexample = "I \twait"
println(tabexample)
```

Here is a combination of these two escape sequences, showing exactly how the tabbing works:

```
tabstops = "12345678901234567890\nI \twait"
println(tabstops)
```

Clearly, the tabbed spaces are eight characters wide. The shift is to the next available starting point:

```
tabstops = "123456789012345678901234567890\nI will just sit here and \t wait"
println(tabstops)
```

Also if several tabs are inserted, all of them make a shift:

```
tabstops = "123456789012345678901234567890\nI just sit \t\tand wait"
println(tabstops)
```

Escaping the escape: `\\`

Of course, if you want `\` to appear in your string, you must escape it:

```
backslash_eg ="1 backslash 4 is 1\\4"
println(backslash_eg)
```

Review and summary

* The characters `\` and `$` are special and don't appear in a formatted string if you simply type them in.
* To see " and \ in a quoted string, escape them as follows: `\"` and `\\`
* To insert a linebreak in a formatted string, use the escape sequence `\n`
* To insert a tab stop (8 spaces on from the previous stop), use the escape sequence `\t`

[2]Strings can be arbitrarily long—a whole novel, linebreaks and all, could be a single string.

Lesson 7, Week 1: Strings III (string interpolation)

AIM

— To explain what string interpolation is, how it works and what it's for.

— To give examples of some of the main uses.

After this lesson, you will be able to

* Explain what string interpolation is and what is for.
* Use the `$()` syntax to do string interpolation.
* Explain what inputs can be used for string interpolation, and give some examples of such inputs.

What is string interpolation, and what is it for?

String interpolation is a method for adding something to a string literal. Why not just make the literal and be done? There are several reasons. One of them is to make a table of values that you have computed. Very often, you want to use formatted strings in the table. But you don't have the values until they are computed. So the code tells Julia to make the formatted string in two steps: first make the part that doesn't require the computed value, and then when the value is available interpolate it. Another is to make labels. Often one needs to label many things with labels that change only a little each time—interpolation is ideal for just adding the changed bit each time.

The `$()` syntax

The `$()` is very simple: you just put it in a literal string. For example, if you had the string `"zxywvut"` and you wanted to interpolate the cry `Help!` in place of the `w`, you could just use `"zxy$("Help!")vut"`.

Important point: the input to `$()` (that is, the inside of the parentheses) must be either a string value, as here, or something that can be formatted as a string value.

Using a variable as input to string interpolation

As we saw, string interpolation is most useful when it uses values not available at the time of writing the code. Here's one that just gives the value of a variable[1]:
`"The value of trackedvariable is $(trackedvariable)"`.

DEMO: using this, for sevaral values of `trackedvariable`.

For example, suppose somehow the variable `todaysdate` was a string with the correct date for today. String interpolation can then be used to label a prediction temperature with today's date.

Shortcut: in some cases, you can omit the parentheses around the input to `$()`. For example
`"The value of trackedvariable is $trackedvariable"`
Why? Well, in the full `$()` syntax, the parentheses are there to delimit exactly the input to be interpolated. So in cases like this, where the `$` character is unambiguously followed by a variable name, Julia decides simply to use that variable as the input for the interpolation. That is, when the input to be interpolated is nothing but a variable name, you can leave off the parentheses[2].

Using an expression as input to string interpolation

Any expression that can be formatted as a string can be the input. This includes characters by themselves and in expressions, the operators `*` and `^` and the shift arithmetic on characters we saw in the previous lesson.

DEMO: `temp = "aa", n = '1'; "the value is $(temp*n)` (note that `n` has a character value—numerical value for `n` doesn't work[3]), this is because of how `*` works.

Because simple numbers so often occur, they can be interpolated, as in `"a$(22)b"` and `"a$(2.222)b"`. This can even include some arithmetic, as in `x=3; "a$(x+39)b"`

[1] Useful to keep track of a computation!

[2] Provided that the variable name is unambiguous—for example, followed by a space or a comma or a full stop. But following it with an exclamation mark would not work. An exclamation mark can be part of a valid name in Julia, so there are two possible names here, the one with and the one without the exclamation mark. Ambiguous!

[3] We'll see later how to get the string version of a numerical value.

Including the character `'$'` in a string literal

Escape it! For example (we must use `print` or `println` — recall that the escaping is only visible after the string is formatted):

```
println("That'll be \$4.99, please")
```

Review and summary

* A literal string can include interpolations via the `$()` syntax
* The input to `$()` is anything that can be formatted as a string: characters, concatenations, character arithmetic, numbers, and number arithmetic
* Interpolations are particularly useful for making strings with values that only become known after the code for making the string was written.
* To include the character value `'$'` in a string value, escape it with `\$`.

Lesson 8, Week 1: Data Containers I (strings)

AIM

— To introduce the idea of a data container, using strings as the example

— To introduce arrays as data containers

— To explain comprehensions as a way to make an array

After this lesson, you will be able to

* Say what a data container is, in general terms
* Use indexing to extract characters from strings
* Use a range as index to extract a string from a string
* Explain what it means for a character to use more than one code unit
* Explain the difference between the functions `length` and `ncodeunits` as applied to strings
* Use a comprehension to write the characters of a string to an array
* Say what an array is, in general terms

Strings as data containers

What is a container? You'll see this word quite often if you ever start reading the Julia documentation. "Container" is a slightly vague term for anything that may contain more than one value. But of course a container is itself a value, so a good way to define them is "a data container is a value that may contain more than one value". Strings are our first example: a string value can contain more than one character value[1].

By the way, containers are absolutely essential when it comes to organising the data on computers. All high-level computer languages[2] offer the programmer several kinds of containers.

[1] Recall that an empty string is valid in Julia, but an empty character is not. Essentially, this is because strings are containers and characters are not.

[2] "High-level" doesn't mean they're special, better than others. In computer science, low-level operations are those that are very close to the fine detail of what actually happens inside a computer. A low-level language is one that deals only or almost only with low-level operations. A high level language is basically a way to translate human ideas into low-level language.

By seeing a string as a container, you realise that you need ways to get the data into the container—everything you've so far learnt about strings is about that. However, you also need ways to get data out of the container, and this lesson is about that.

Indexing into strings

Indexing is one way to get data out of a string[3]. The idea is that in a string there's a first character, a second character, a third, and so on. This can function in a way similar to house numbers in a street: `eg1 = "abcde"; eg1[1]`

Point 1: you tell Julia the value of the index by delimiting with square brackets. Point 2: in this case, indexing works perfectly, that is, `eg1[k]` with give us the `k`-th character in `eg1`. Provided of course that the value of `k` is one of `1, 2, 3, 4, 5`; anything else throws an error.

Let me emphasise again the use square brackets in this case, as opposed to the round parentheses you've seen before. In Julia, it is very important to use the correct delimiters!

You can index into a string directly: `"a1b2"[3]`.

Finally, indexing can be to part of a string (pay careful attention to how the colon is used)
`greeting = "Hello, world"; greeting[2:8]`

The code fragment is `2:8` here is a range[4]. It starts with the index value 2 and ends with the index value 8. You can use variables to construct a range, assign the result to a variable, and use that variable to index a string:
`init, last = 1, 3; myrange = init:last; "1a2b3c"[myrange]`

A range can use steps[5] bigger than 1: `greeting[3:2:7]`, which extracts the 3rd, 5th and 7th characters. You can even use negative steps, as in reversing the order: `greeting[end:-1:1]`.

Note that the result of indexing with a range is always a string, even if the range starts and ends with the same value: `greeting[2:2]`

In other words, to get a single character out of a string, you use the correct index, not a range. To get that same character as a string, index with a range that starts and ends with the same index value.

[3] All of the data you get out will consist of characters, of course, because that is the only kind of data inside a string.

[4] Range one of several ways to index to several values at the same time. In this course, we do not explore the topic of indexing into containers in any depth at all. However, expert use of Julia requires a good knowledge of all the possibilities.

[5] Many people use the word *stride* for the step length in a range

A complication: the difference between `length` and `ncodepoints`

Sadly, not all strings index as nicely as `eg1`. That is Julia uses characters with variable width[6]—some take more memory to store than others. In Julia, this is expressed in terms of the number of code units the character requires for storing in memory. Each character in `eg1` requires just one code unit. The function `length`, when applied to a strings, returns the number of characters in the string. The function `ncodeunits` returns the number of code units.

Because `eg1` uses characters which are a one code unit wide, both `length(eg1)` and `ncodeunits(eg1)` return the value `5`.

The Greek alphabet is an example of where they differ [DEMO reminder: how to enter Greek letters; `length("α")` vs `ncodeunits(α)`]

This means that `"α"[1]` is valid code, but `"α"[2]` not. [DEMO: this is not a bounds error].

Similarly, for `eg2 = "αβ"` we have that `eg2[1]` and `eg2[3]` are valid code, but `eg2[2]` is not, nor is `eg2[4]`. The rule is: if the index points to the first code unit of a character, the whole character is returned. If it points to any other code unit, an error is thrown.

Moreover, if a range starts and ends with an index that corresponds to a character, that index is valid. Here, `eg2[1]` and `eg2[3]` are valid and hence so is `eg2[1:3]`.

For completeness: `ncodeunits` can take a single character as input and give its width. [DEMO]

Solution: use a comprehension to make an array of the characters

If you need the characters of a string separately, you can make an array of them. In that case, the variable width doesn't matter. There are many ways to do so, but using a comprehension is by far the most convenient:
`greeting_array = [char for char in greeting]`.
This is an assignment, i.e. a left hand side name linked by an equals sign to a right hand side value.

The right hand side is your first bit of tricky code. It is worth paying it very careful attention. There are a whole lot of technical issues to deal with:

First Please be aware the the word "Array" is a technical term in Julia, with a very precise meaning. Like a string, it is a container in which elements can be accessed by means of indexing. It is different from a string in two ways: you can change the values inside an array, and any kind of value can be used, whereas a string can only contain valid characters.

[6] So do other languages, but they may handle the issue differently. There is no way around it, really. Computers these days are dominated by variable-width encodings of characters. You can read more about this in the Julia documentation, where there is also a brief discussion of the reasons in favour of doing it the Julian way.

Second The square brackets as delimiters is how we know that we are telling Julia to create an array.

Third The name `char` inside the square brackets occurs twice because it has two different tasks.

Fourth The "for" in the square brackets is a reserved keyword[7]. This means that it cannot be used as a name, despite being validly formed.

Fifth The `in` in the square brackets is an operator whose meaning we explain below. It is not reserved, so it can be used as a name[8].

That expression, from the opening square bracket to the closing square bracket, is a comprehension. The square brackets indicate that the value will be an array. The name `char` is purely temporary, and doesn't continue to exist after the assignment is completed.

How does the comprehension work? It is quite simple: the second `char` is a variable that takes the values in `greeting` one after the other. We say that it iterates[9] over the values in `greeting`. The first `char` creates the value that is written into the array. Once the iteration is over, the resultant array is assigned to `greeting_array`. [DEMO: create other values to write into the array]

Question: how does Julia know to iterate inside a comprehension? Answer: the combination of the keyword `for` and the operator `in`. So this combination is the only way to create a comprehension, and it only works inside square brackets, because comprehensions are used only to make arrays[10].

Let's summarise: the syntax of a comprehension is `[<value> for <dummyvar> in <container>]`. Here, `<container>` indicates the container you iterate over, `<dummyvar>` indicates that you need a variable to hold each value one by one as it is extracted, and `<value>` indicates the expression used to create the values that goes into the array. The square brackets, the `for` and the `in` are always used in the same way to make a comprehension.

A few more introductory remarks about arrays

Arrays are a very efficient way to store memory in a computer, particularly when the elements of the array have fixed width[11].

Arrays like `greeting_array` are said to be one-dimensional, because they're like a street, you need only one number to index a value. But often it is convenient to use more dimensions, for example an appointment for nine o'clock on 13 March has three dimensions: the month, the day and the time. In some computer applications, the number of dimensions gets very large, but many of the elements are not used. Such an array is said to be sparse[12], and efficiently working with sparse arrays is one of Julia's great strengths.

[7] You can see the list of reserved keywords at `https://docs.julialang.org/en/v1/base/base/#Keywords-1`

[8] It might even be a good name in some contexts, but short names like this should be avoided, unless you have clear and good reason for using them.

[9] Iteration is very important in computer programs, and we will see it again.

[10] This is typical of Julia: specialised and very compact code to create a frequently used container. Almost all computer languages have ways to write specialised and compact code.

[11] This means that an array of characters is not actually the most efficient way to store a string, which is why Julia has a separate way to store strings.

[12] Storing one's appointments for a whole year in a three-dimensional array is almost certain to result in a very sparse array!

We will not on this course go into the details of how to use arrays efficiently in Julia. This is partly because it is an advanced topic, but mostly because Julia provides a very large number of ways to deal with arrays, each of which applies only to a few specialised cases. That is, for a particular application, it is worth learning efficient array handling. However, it is quite unpredictable whether in the next application you take on, efficient array handling will work the same way. This is one of those areas in programming where we believe it is not helpful to learn everything[13].

Review and summary

This has been your longest and most technical lecture so far, with several new ideas. Please make sure that you go through exercises. If you cope with them you are well on your way to being a Julia programmer!

* Strings and arrays are data containers
* The data in a string can be extracted using indexing using a pair of square brackets as the delimiters.
* Indexing into a string can a single integer or a range
* A range index into a string returns another string
* The width of a character is the number of code units used to store that character
* Characters in Julia have variable width; use the function `ncodeunits` to determine width
* `length` returns the number of characters in a string; `ncodeunits` returns the width
* A comprehension creates an array by iterating over a container
* A comprehension consist of square brackets around an iteration
* The iteration in a comprehension uses the reserved keyword `for` and the operator `in`
* The syntax of a comprehension is `<value> for <dummyvar> in <container>`
* Arrays in general are an advanced topic we do not take on in this course

[13] Unless you want to specialise in array handling, of course!

Lesson 9, Week 1: Functions I

Functions in Julia is a very big subject, and in this first lecture we'll cover only a very little of it. In fact, we'll discuss only two built-in functions, but in the process we will consider multiple inputs, keywords and defaults. We will also start on the topic of inputs that differ by type[1], but only to a limited extent, reserving a more complete discussion for later in the course.

We will also introduce you to the the `?` way of using the REPL to query Julia's help system. You should plan to use this system early and often[2]!

AIM

— To learn to use the functions `string` and `join`

— To understand the difference multiple inputs make to `join`

— To understand the difference that keywords make to `string` and the role of defaults

After this lesson, you will be able to

* Use `string` to combine several values into one string
* Use `string` to convert an integer into a string (optionally specifying a the base to use for the conversion)
* Use `join` to make a single string from a 1-dimensional array of strings (optionally specifying separators)

`string` as method for combining values into a single string

We haven't seen multiple inputs to a function yet, here is the first example. Note that the comma is used as the delimiter—that is, the character that separates values from each other.

DEMO cases involving strings, characters and numbers, including variables with those as values, escape sequences, and `print`ing them.

[1] The word "type" has a Julian meaning that in this lecture we indicate very roughly. In a later lecture we consider Julia's type system in more detail.

[2] Until you don't need it, of course

Calling `string` with an integer value allows some optional formatting

Let's turn to Julia's help system: simply enter `?` at the `julia>` prompt in the REPL. Notice that now the you get the `help?>` prompt. To get information, type a topic and hit Enter. [DEMO: help on `string`]

Interesting. It turns out that `string` does different things! Question: how does Julia decide? Anser: the pattern of inputs. In particular, whether or not you specify one or both of `base` and `pad`.

[DEMO: various possibilities with `base`, making sure to show some error messages]

We show this here to emphasise the use of the keyword `base` here. A keyword argument to a function is optional: if you say nothing, then Julia will use a default value, that is, Julia will act as if you gave the input `base=10`.

Now for the keyword `pad`. [DEMO]

`join` as a method for combining strings, with optional formatting

In the preceding sections we saw a function that takes multiple separate inputs. But you could put values into a container and just pass that container to a function as a single value. Let's use `?` again to get help.

The following should work:

```
strarray = ["a", "bb", "ccc", "dddd"]
join(strarray, "; ", " but not ")
```

DEMO cases where not all the values are srings, some are characters and/or numbers

But there's more: the help tex on `join` has square brackets arount the last argument in the function call. This is means that it is optional—it can be left out. [DEMO]

Keywords defaults vs optionally including some variables

Notice how `join` just concatenates an array of string values into a single string with no extra characters *unless* you pass it two or three arguments. That is, `join` behaves differently depending on the pattern of inputs. On the other hand, `string` changes behaviour when we include at least one of the keywords `base` or `pad`. In both cases, though, the behaviour you get depends on the pattern of inputs you supply. We will see this kind of thing again and again: most functions in Julia are generic,

in the sense that they actually consist of many methods[3]. Which method you actually get when you call the function depends on the pattern of inputs you give.

DEMO: `methods(join)` reveals six methods, with six different patterns of input.

We will discuss generic functions and their patterns of input in more detail later in the course.

But why the difference? Couldn't `string` do things in the same way as `join`? Perhaps, but there is one obvious advantage that keywords have: if you omit some of them, the others have default values that are used. But `join`, because it doesn't use keywords with default values, doesn't allow us to omit the first delimiter and use only the second one.

For example, `join(strarray, , " and at last, ")` throws an error. We have to insert something as the first delimiter, even if it is just the empty string, in order to be able to set the second delimiter.

When you write your own functions, you may have to decide whether or not to use keywords. How to make the best decision is a relatively advanced topic we do not address in this course.

Review and summary

* The function `string` when acting on many inputs of all kinds (separated by commas) concatenates them into a single string
* The function `string` can operate on just a single integer, where the keywords `base` and `pad` allow formatting to other than decimal numbers.
* The function `join` concatenates an array of strings into a single string
* The function `join` has alternate methods which allow delimiters to be inserted between the strings it concatenates, depending on how many values you put into its input list.

[3]From the point of view of computer science, this is one of the most interesting and unusual aspects of the Julia language, and many would say also one of its greatest strengths.