Preface

In this book, we develop the mathematical ideas underlying the modern practice of data science. The goal is to do so accessibly and with a minimum of prerequisites.* For example, we will start by reviewing sets and functions from a data-oriented perspective. On the other hand, we will take a problem-solving approach to the material and will not shy away from challenging concepts. To get the most out of the course, you should prepare to invest a substantial amount of time on the exercises.

* Experience with basic calculus is necessary, and some prior experience with linear algebra and programming will be helpful.

This text was originally written to accompany the master's course DATA 1010 at Brown University. The content of this PDF, along with hints and solutions for the exercises in this book, will be available on the edX platform starting in the fall semester of 2019, thanks to the BrownX project and the Brown University School of Professional Studies.

The author would like to acknowledge Isaac Solomon, Elvis Nunez, and Thabo Samakhoana for their contributions during the development of the DATA 1010 curriculum. They wrote some of the exercises and many of the solutions that appear in the BrownX course.

Contents

1 Programming in Julia

In this course, we will develop mathematical ideas in concert with corresponding computational skills. This relationship is symbiotic: writing programs is an important ingredient for applying mathematical ideas to real-world problems, but it also helps us explore and visualize math ideas in ways that go beyond what we could achieve with pen, paper, and imagination.

We will use *Julia*. This is a relatively new entrant to the scientific computing scene, having been introduced publicly in 2012 and reaching its first stable release in August of 2018. Julia is ideally suited to the purposes of this course:

1. **Julia is designed for scientific computing**. The way that code is written in Julia is influenced heavily by its primary intended application as a scientific computing environment. This means that our code will be succinct and will often look very similar to the corresponding math notation.
2. **Julia has benefits as an instructional language**. Julia provides tools for inspecting how numbers and other data structures are stored internally, and it also makes it easy to see how the built-in functions work.
3. **Julia is simple yet fast**. Hand-coded algorithms are generally much faster in Julia than in other user-friendly languages like Python or R. This is not always important in practice, because you can usually use fast code written by other people for the most performance-sensitive parts of your program. But when you're learning fundamental ideas, it's very helpful to be able to write out simple algorithms by hand and examine their behavior on large or small inputs.
4. **Julia is integrated in the broader ecosystem**. Julia has excellent tools for interfacing with other languages like C, C++, Python, and R, so can take advantage of the mountain of scientific computing resources developed over the last several decades. (Conversely, if you're working in a Python or R environment in the future, you can write some Julia code and integrate it into your Python or R program).

1.1 Environment and workflow

There are four ways of interacting with Julia:

1. *REPL*. Launch a read-eval-print loop from the command line. Any code you enter will be executed immediately, and any values returned by your code will be displayed.
2. *Script*. Save your code in a file called `example.jl`, and run `julia example.jl` the command line to execute all the code in the file.
3. *Notebook*. Like a REPL, but allows inserting text and math expressions for explanation, grouping code into blocks, multimedia output, and other features. The main notebook app is called **Jupyter**, and it supports Julia, Python, R and many other languages.
4. *IDE*. An integrated development environment is a program for editing your scripts which provides various code-aware conveniences (like autocompletion, highlighting, and many others). **Juno** is an IDE for Julia.

Some important tips for getting help as you learn:

1. Julia's official documentation is available at `https://docs.julialang.org` and is excellent. The learning experience you will get in this chapter is intended to get you up and running with the ideas we will use in this course, but if you do want to learn the language more extensively, I recommend investigating the resources linked on the official Julia learning page: `https://julialang.org/learning/`
2. You can get help within a Julia session by typing a question mark before the name of a function whose documentation you want to see.
3. Similarly, `apropos("eigenvalue")` returns a list of functions whose documentation mentions "eigenvalue"

Exercise 1.1.1

Install Julia 1.0 on your system, start a command-line session by running `julia` from a terminal or command prompt, What does the ASCII-art banner look like (the one that pops up when you start a command line session)?

1.2 Fundamentals

We begin by developing some basic vocabulary for the elements of a program.

1.2.1 Variables and values

A **value** is a fundamental entity that may be manipulated by a program. Values have **types**; for example, `5` is an `Int` and `"Hello world!"` is a `String`. Types are important for the computer to keep track of, since values are stored differently depending on their type. You can check the type of a value using the `typeof` function: `typeof("hello")` returns `String`.

A **variable** is a name used to refer to a value. We can **assign** a value (say `41`) to a variable (say `age`) as follows:

```
age = 41
```

A **function** performs a particular task. For example, `print(x)` writes a string representation of the value of the variable `x` to the screen. Prompting a function to perform its task is referred to as **calling** the function. Functions are called using parentheses following the function's name, and values needed by the function are supplied between these parentheses.

An **operator** is a special kind of function that can be called in a special way. For example, the multiplication operator `*` is called using the mathematically familiar *infix notation* `3 * 5`, or `*(3,5)`.

A **statement** is an instruction to be executed. For example, the assignment `age = 41` is a statement. An **expression** is a combination of values, variables, operators, and function calls that a language interprets and **evaluates** to a value. For example, `1 + age + abs(3*-4)` is an expression which evaluates* to `54`.

`abs` is the absolute value function

A sequence of statements or expressions can be collected into a **block**, delimited by `begin` and `end`. The statements and expressions in a block are executed sequentially. If the block concludes with an expression, then the block returns the value of that expression. For example,

```julia
begin
    y = x+1
    y^2
end
```

is equivalent to the expression `(x+1)^2`.

The **keywords** of a language are the words that have special meaning and cannot be used for other purposes. For example, `begin` and `end` are keywords in Julia.

Exercise 1.2.1

What value does the following expression evaluate to? What is the type of the resulting value? Explain.

```julia
begin
    x = 14
    x = x / 2
    y = x + 1
    y = y + x
    2y + 1
end
```

1.2.2 Basic data types

Julia, like most programming environments, has built-in types for handling common data like numbers and text.

1. Numbers.
   (a) A numerical value can be either an **integer** or a **float**. The most commonly used integer and floating point types are `Int64` and `Float64`, so named because they are stored using 64 bits. We can represent integers exactly, while storing a real number as a float often requires slightly rounding it (see the *Numerical Computation* chapter for details). A number is stored as a float or integer according to whether it contains a decimal point, so if you want the value 6 to be stored as a float, you should write it as `6.0`.
   (b) Basic arithmetic follows order of operations: `1 + 3*5*(-1 + 2)` evaluates to `16`.
   (c) The exponentiation operator is `^`.
   (d) Values can be compared using the operators* `==`,`>`,`<`,`≤`,`≥`.
   (e) Julia integers will overflow unless you make them big[†] , so `big(2)^100` works, but `2^100` evaluates to `0`.
2. Strings
   (a) Textual data is represented in a **string**. We can create a string value by enclosing the desired string in quotation marks: `a = "this is a string"`.

* For the last two symbols, use `\le«tab»` and `\ge«tab»`

† Operations with big-type numbers are much slower than operations with usual ints and floats

(b) We can find the number of characters in a string with the `length` function: `length("hello")` returns `5`.

(c) We can combine two strings with the asterisk operator (`*`): `"Hello " * "World"`

(d) We can insert the value of a variable into a string using *string interpolation*:

```julia
x = 4.2
print("The value of x is $x")
```

3. Booleans

   (a) A **boolean** is a special data type which is either `true` or `false`

   (b) The basic operators that can be used to combine boolean values are

and	`&&`
or	`||`
not	`!`

Exercise 1.2.2

(i) Write a Julia expression which computes $\frac{1}{a+\frac{2}{3}}$ where a is equal to the number of characters in the string `"The quick brown fox jumped over the lazy dog"`

(ii) Does Julia convert types when doing equality comparison? In other words, does `1 == 1.0` return `true` or `false`?

Here's an example showing how to incorporate Julia output into your document:

```julia
a = 2
```

The value of `a^200` is 0, but the value of `big(a)^200` is

$$1606938044258990275541962092341162602522202993782792835301376.$$