1 Overview of Julia commands

Julia is a programming language that is freely available.

Launch Binder

We can use `Julia` without installation by clicking the "launch Binder" badge. The notebook `CalculusWithJulia.ipynb` is a blank notebook except for the command to load the `CalculusWithJulia` package.

Alternatively, the juliabox.com service provides access to `Julia` through the internet, though the free access is slated to be discontinued. Both use the Jupyter notebook interface, which we will assume, though many other means of interacting with `Julia` are available.

1.1 Commands

Commands are typed into a notebook cell in `Jupyter`. (Or at the command line.)

```
2 + 2   # use shift-enter to evaluate
```

4

Commands are executed by using `shift-enter` or the play button in `Jupyter`.

Commands may be separated by new lines or semicolons, allowing multiple commands per cell.

On a line, anything after a `#` is a *comment*.

The results of the last line executed will be displayed in an output area. Separating values by commas allows more than one value to be displayed. Explicit printing can be done to output intermediate values.

1.2 Numbers, variable types

`Julia` has many different number types beyond the floating point type employed by most calculators. These include

- Floating point numbers: `0.5`
- Integers: `2`
- Rational numbers: `1//2`
- Complex numbers `2 + 0im`

As much as possible, operations involving certain types of numbers will produce output of a given type. For example, both of these divisions produce a floating point answer, even though mathematically, they need not:

```
2/1, 1/2
```

```
(2.0, 0.5)
```

Some operations won't work with integer types, but will with floating point types, as the type of output can't be assured.

Powers with negative bases, like `(-3.0)^(1/3)`, are not defined. However, `Julia` provides the special-case function `cbrt` (and `sqrt`) for handling these.

Integer operations may silently overflow, producing odd answers, at first glance:

```
2^64
```

0

(Though the output is predictable, if overflow is taken into consideration appropriateley.)

When different types of numbers are mixed, `Julia` will usually promote the values to a common type before the operation:

```
(2 + 1//2) + 0.5
```

3 . 0

`Julia` will first add `2` and `1//2` converting `2` to rational before doing so. Then add the result, `5//2` to `0.5` by promoting `5//2` to the floating point number `2.5` before proceeding.

`Julia` uses a special type to store a handful of constants, of which both `pi` and `e` are used here. Theh special type allows these to be treated without round off, until they mix with other floating point numbers. There are some functions that require these be explicitly promoted to floating point. This can be done by calling `float`.

The standard mathematical operations are implemented by `+`, `-`, `*`, `/`, `^`. Parentheses are used for grouping.

1.2.1 Vectors

A vector is an indexed collection of similarly typed values. Vectors can be constructed with square brackets (syntax for concatenation):

```
[1,1,2,3,5,8]
```

```
6-element Array{Int64,1}:
 1
 1
 2
 3
 5
 8
```

(Vectors are used as a return type so some familiarity is needed.)

Regular arithmetic sequences can be defined by either:

- Range operations: `a:h:b` or `a:b` which produces a generator of values starting at `a` separated by `h` (`h` is `1` in the last form) until they reach `b`.
- `range(a, b, length=n)` which produces a generator of `n` values between `a` and `b`; (`range(a, stop=b, length=n)` is needed for version `v1.0.0` of 'Julia.)

These constructs return range objects. A range object *compactly* stores the values it references. To see all the values, they can be collected with the `collect` function, though this is hardly needed in practice.

1.3 Variables

Values can be assigned variable names, with `=`. There are some variants

```
x = 2
a_really_long_name = 3
a, b = 1, 2    # multiple assignment
a1 = a2 = 0    # chained assignment, sets a2 and a1 to 0
```

0

The names can be short, as above, or more verbose. They can't start with a number, but can include numbers. It can also be a fancy unicode or even an emoji.

We can then use the variables to reference the values:

```
x + a_really_long_name + a - b
```

4

Names may be repurposed, even with values of different types (a dynamic language), save for function names, which have some special rules and can only be redefined as an another function. (Generic functions are central to `Julia`'s design. Generic functions use a method table to dispatch on, so once a name is assigned to a generic function, it can not be used as a variable name; the reverse is also true.

1.4 Functions

Functions in `Julia` are just regular objects. In these notes, we often pass them as arguments to other functions. There are many built-in functions and it is easy to define new functions.

We "call" a function by passing argument(s) to it, grouped by parentheses:

```
sqrt(10)
sin(pi/3)
log(5, 100)   # log base 5 of 100
```

2 . 8 6 1 3 5 3 1 1 6 1 4 6 7 8 6 7

With out parentheses, the name (usually) refers to a generic name and the output lists the number of available implementations.

```
log
```

```
log (generic function with 48 methods)
```

1.4.1 Built-in functions

`Julia` has numerous built-in mathematical functions, we review a few here:

Powers logs and roots Besides `^`, there are `sqrt` and `cbrt` for powers. In addition basic functions for exponential and logarithmic functions:

```
sqrt(x), cbrt(x)
exp(x)
log(x) # base e
log10(x), log2(x), log(b, x)
```

Trigonometric functions The 6 standard trig functions are implemented; their implementation for degree arguments; their inverse functions; and the hyperbolic analogs.

```
sin, cos, tan, csc, sec, cot
asin, acos, atan, acsc, asec, acot
sinh, cosh, tanh, csch, sech, coth
asinh, acosh, atanh, acsch, asech, acoth
```

If degrees are preferred, the following are defined to work with degrees:

```
sind, cosd, tand, cscd, secd, cotd
```

Useful functions Other useful and familiar functions are defined:

- `abs(x)`: absolute value
- `sign(x)`: is $|x|/x$ except at $x = 0$, where it is 0.
- `floor(x)`, `ceil(x)`: greatest integer less or least integer greater
- `max(a,b)`, `min(a,b)`: larger (or smaller) of `a` or `b`
- `maximum(xs)`, `minimum(xs)`: largest or smallest of the collection referred to by `xs`

1.4.2 User-defined functions

Simple mathematical functions can be defined using standard mathematical notation:

```
f(x) = -16x^2 + 100x + 2
```

```
f (generic function with 1 method)
```

The argument `x` is passed into the body of function.

Other values are found from the environment where defined:

```
a = 1
f(x) = 2*a + x
f(3)   # 2 * 1 + 3
a = 4
f(3)  # now 2 * 4 + 3
```

11

User defined functions can have 0, 1 or more arguments:

```
area(w, h) = w*h
```

```
area (generic function with 1 method)
```

Julia makes different *methods* for *generic* function names, so functions whose argument specification is different are different functions, even if the name is the same. This is *polymorphism*. The practical use is that it means users need only remember a much smaller set of function names.

Functions can be defined with *keyword* arguments that may have defaults specified:

```
f(x; m=1, b=0) = m*x + b      # note ";"
f(1)                          # uses m=1, b=0   -> 1 * 1 + 0
f(1, m=10)                    # uses m=10, b=0  -> 10 * 1 + 0
f(1, m=10, b=5)               # uses m=10, b=5  -> 10 * 1 + 5
```

15

Longer functions can be defined using the `function` keyword, the last command executed is returned:

```
function f(x)
  y = x^2
  z = y - 3
  z
end
```

```
f (generic function with 1 method)
```

Functions without names, *anonymous functions*, are made with the `->` syntax as in:

```
x -> cos(x)^2 - cos(2x)
```

```
#2 (generic function with 1 method)
```

These are useful when passing a function to another function or when writing a function that *returns* a function.

1.5 Conditional statements

`Julia` provides the traditional `if-else-end` statements, but more conveniently has a `ternary` operator for the simplest case:

```
our_abs(x) = (x < 0) ? -x : x
```

```
our_abs (generic function with 1 method)
```

1.6 Looping

Iterating over a collection can be done with the traditional `for` loop. However, there are list comprehensions to mimic the definition of a set:

```
[x^2 for x in 1:10]
```

```
10-element Array{Int64,1}:
   1
   4
   9
  16
  25
  36
  49
  64
  81
 100
```

1.7 Broadcasting, mapping

A function can be applied to each element of a vector through mapping or broadcasting. The latter is implemented in a succinct notation. Calling a function with a "." before its opening "(‘ will apply the function to each individual value in the argument:

```
xs = [1,2,3,4,5]
sin.(xs)     # gives back [sin(1), sin(2), sin(3), sin(4), sin(5)]
```

```
5-element Array{Float64,1}:
  0.8414709848078965
  0.9092974268256817
  0.1411200080598672
 -0.7568024953079282
 -0.9589242746631385
```

1.8 Plotting

Plotting is *not* built-in to `Julia`, rather added through add-on packages. `Julia`'s `Plots` package is an interface to several plotting packages. We mention `plotly` (built-in) for web based graphics, and `gr` for other graphics.

To use an add-on package, it must have been installed and it must be loaded each session. In these notes, the necessary packages are all loaded when an accompanying package, `CalculusWithJulia` is loaded. Assuming it has been installed, this command will do so:

```
using CalculusWithJulia
```

With `Plots` loaded, we can plot a function by passing the function object by name to `plot`, specifying the range of x values to show, as follows:

```
plot(sin, 0, 2pi) # plot a function - by name - over an interval [a,b]
```

```
Plot{Plots.PlotlyBackend() n=1}
```

This is in the form of **the** basic pattern employed: `verb(function_object, arguments...)`. The verb in this example is `plot`, the object `sin`, the arguments `0, 2pi` to specify `[a,b]` domain to plot over.

Plotting more than one function over $[a, b]$ is achieved through the `plot!` function, which modifies the existing plot (`plot` creates a new one):

```
plot(sin, 0, 2pi)
plot!(cos, 0, 2pi)
plot!(zero, 0, 2pi)
```

```
Plot{Plots.PlotlyBackend() n=3}
```

Plotting an *anonymous* function is a bit more immediate:

```
plot( x -> exp(-x/pi) * sin(x), 0, 2pi)
```

```
Plot{Plots.PlotlyBackend() n=1}
```

The `Plots` package has other types of plots. Of note is `scatter` which is used to make a scatter plot of two data sets.

1.9 Equations

Notation for `Julia` and math is *similar* for functions - but not for equations. In math, an equation might look like:

$$x^2 + y^2 = 3$$

In `Julia` the equals sign is **only** for *assignment*. The *left-hand* side of an equals sign in `Julia` is reserved for a) variable assignment; b) function defintion (via `f(x) = ...`); and c) indexed assignment to a vector or array. (Vectors are indexed by a number allow retrieval and setting of the stored value in the container. The notation mentioned here would be `xs[2] = 3` to assign to the 2nd element a value `3`.

1.10 Symbolic math

Symbolic math is available through an add-on package `SymPy`. As with `Plots`, this package is also loaded with `CalculusWithJulia`. Once loaded, symbolic variables are created with `@vars`:

```
using SymPy   # load package, not needed here as it was done already with
CalculusWithJulia
@vars x a b c # no commas here, though `@vars(x,a,b,c)` can be used
```

```
(x, a, b, c)
```

Symbolic expressions - unlike numeric expressions - are not immediately evaluated, though they are simplified:

```
p = a*x^2 + b*x + c
```

$$ax^2 + bx + c$$

To substitute a value, we can use `pair` notation (`variable=>value`):

```
p(x=>2), p(x=>2, a=>3, b=>4, c=>1)
```

```
(4*a + 2*b + c, 21)
```

This is convenient notation for calling the `subs` function.

SymPy expressions of a single free variable can be plotted directly:

```
plot(64 - (1/2)*32 * x^2, 0, 2)
```

```
Plot{Plots.PlotlyBackend() n=1}
```

SymPy has functions for manipulating expressions: `simplify`, `expand`, `together`, `factor`, `cancel`, `apart`, `args`, ...

SymPy has functions for basic math: `factor`, `solve`, ...

SymPy has functions for calculus: `limit`, `diff`, `integrate`