

# *Mathematica*: Hands on with the basics

This worksheet describes a few very basic concepts of *Mathematica*. There are a few pretty easy exercises. Do 'em.

## Arithmetic

Of course, *Mathematica* can add, subtract, multiply, divide. For example,

```
(E + Pi ^ 2) / 42
```

Note that the result is an *exact* number. You may get decimal approximation as follows.

```
% // N
```

Note that the % stands for the previous output and //N passes that result to the numerical evaluator. You can get 1000 digits of  $\pi$  quite easily.

```
N[Pi, 1000]
```

You should understand that there is a huge difference between *exact* and *approximate* numbers. Computations are typically much faster with approximate number, but they're, well, approximations.

**Exercise 1:** Compute an approximation of  $e^{\pi\sqrt{163}}$  to 29 digits of precision.

## Brackets, braces, and parentheses

These all have distinct purposes in *Mathematica*. Brackets [] are used to enclose the argument of a function such as:

```
N[Tan[1]]
```

Parentheses () are used for grouping in mathematical expressions such as:

```
(2 * 5 - 1) / 3
```

Braces {} are used to form lists. Lists play a very important role in *Mathematica* as they are a fundamental data structure. We will probably devote a week to list manipulation. Many functions can act on lists.

```
N[{Pi, E, Sqrt[2]}]
```

Some functions return lists.

```
digits = IntegerDigits[2^1000]
```

And some functions manipulate lists in interesting ways. Can you see what the following command does?

```
Tally[digits]
```

If you want information on this command, you can type `?Tally`. You can also type `Tally` into the Documentation Center available through the help menu.

### Exercise 2:

- How many digits does  $2^{1000000}$  have? (I'd use the `Length` command in conjunction with `IntegerDigits`.)
- What is the frequency of the least common digit in  $2^{1000000}$ ?

## Lists

Lists are a fundamental data structure in *Mathematica* and much of discrete mathematics in particular involves list manipulation. Lists may be entered as comma separated sequences of expressions enclosed in braces. Here are three examples of lists.

```
{1, 2, 3};  
{a, b, c};  
{{"Mark", "Ann", "Audrey"},  
 {"Gas", "Phone", "Electricity"}};
```

We frequently need to build lists more programmatically. Two fundamental commands to do so are `Table` and `Range`. Here's two ways to generate the first 100 odd primes.

```
Table[Prime[n], {n, 2, 101}]
```

The general syntax is `Table[f[n], {n, start, fin}]`. Make sense? We can verify the length is correct as follows.

```
% // Length
```

An alternative approach is as follows.

```
Prime /@ Range[2, 101]
```

In this example, the `Range` command generates the list  $\{2, 3, 4, \dots, 101\}$  and the `Map` operator (`/@`) maps `Prime` onto the list - a very common construct.

**Exercise 3:** Use the `FactorInteger` command to find the factorizations of all integers between 1000000 and 1001000.

## Variables and functions

We can set a variable using `=` and then refer to it later. For example,

```
a = Range[10];  
a^2
```

Defining functions is similar, although the syntax typically used is a bit different.

```
f[x_] := (x^2 + 2) / (2 x);  
f[2]
```

Note the `:=`, rather than just `=`. The plain old `=` (also called `Set`) evaluates the right side and sets the result immediately, while `:=` (also called `SetDelayed`) evaluates the definition only when the expression is called. These are typically (but not always) the desired behaviors when defining variables and functions respectively. You can see the difference in the following two commands.

```
def1 = RandomReal[];  
Table[def1, {10}]
```

Note that the result of calling `def1` is the same every time, since the result is set immediately. Now try the following:

```
def2 := RandomReal[];  
Table[def2, {10}]
```

Now, the right hand side is re-evaluated with each application of `def2` resulting in list of distinct numbers. We'll use exactly this distinction later, when constructing certain types of trees.

One common way to generate lists using functions is via iteration. The basic built in function for performing iteration is `NestList`. For example, here we the cosine function 20 times from the starting point 1.

```
NestList[Cos, 1.0, 20]
```

You might be interested to see what happens if you start at 1, rather than 1.0.

**Exercise 4:** Iterate the function  $f(x) = (x^2 + 2)/(2x)$  defined above starting from your favorite number. Is there a limit?

## Graphing functions

The main tool for graphing functions is the `Plot` command.

```
Plot[x^2, {x, -2, 2}]
```

We can plot more than one function at once by using a list of functions. For example, the following command illustrates shows the solution to the equation  $\cos(x) = x$ .

```
Plot[{x, Cos[x]}, {x, -1, 2},  
PlotStyle -> {Red, Blue}]
```

This command illustrates our first use of an *option*, namely `PlotStyle->{Red,Blue}` so we can distinguish the plots. There are many graphics options and many graphics commands. We'll look at these in a bit more detail later. For now let's take a look at `ListPlot`, a discrete graphics command. Given a list of numbers, `ListPlot` plots them versus their position on a list. For example, let's generate the first few Fibonacci numbers.

```
fibs = Table[Fibonacci[k], {k, 1, 15}]
```

Now, we `ListPlot` them.

```
ListPlot[fibs]
```

Looks like they grow about exponentially.

**Exercise 5:** Let  $f(x) = x \sin(x^2)$ . Plot the graphs of  $f(x)$  and  $f'(x)$  on the same axes.

Note: You can define `f[x]` and then refer to its derivative by `f'[x]`.

*Mathematica* has a powerful graph visualization tool called `GraphPlot`. We simply generate a list of edges of the form  $v_1 \rightarrow v_2$  and pass the result to `GraphPlot`.

```
G = {1 -> 2, 2 -> 3, 3 -> 4, 4 -> 1};  
GraphPlot[G]
```

The vertices can be arbitrary *Mathematica* expressions. Strings make great labels, for example.

```
G = {"Mark" -> "Ann", "Ann" -> "Audrey", "Audrey" -> "Mark"};  
GraphPlot[G, VertexLabeling -> True]
```

We can also generate graphs programmatically.

```
n = 19;  
G = Table[i -> Mod[i^3 + 1, n], {i, 0, n - 1}];  
GraphPlot[G]
```

**Exercise 6:** Use `GraphPlot` to plot the complete graph on 4 vertices. How might you do the complete graph on 44 vertices?