Automating the Tedious Stuff

(Functional programming and other Mathematica magic)

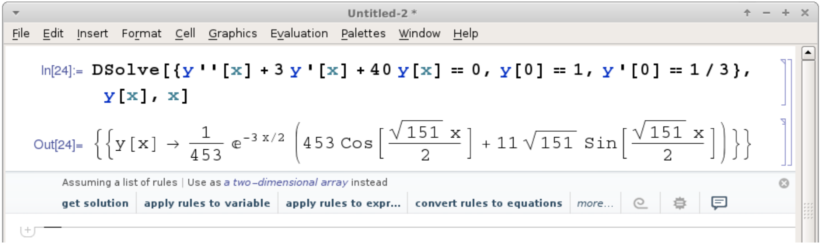
Connor Glosser

Michigan State University
Departments of Physics &
Electrical/Computer Engineering

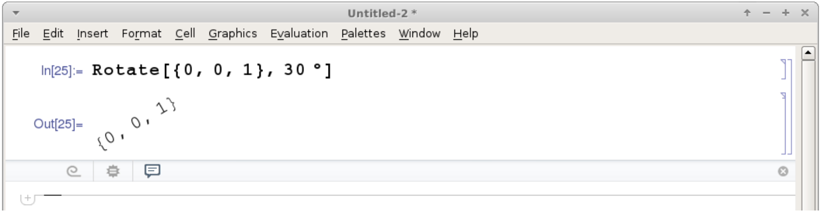
π, 2014

Table of Contents

Mathematica is great. . .



. . . but it's also kind of stupid.



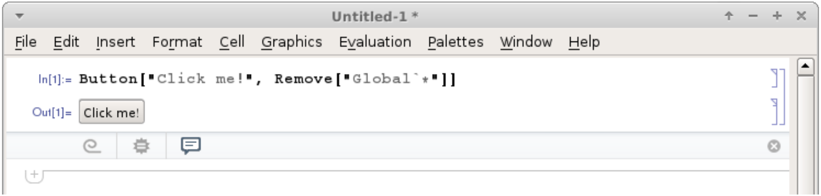
About this talk

"Mr. Osborne, may I be excused? My brain is full."

What this talk is

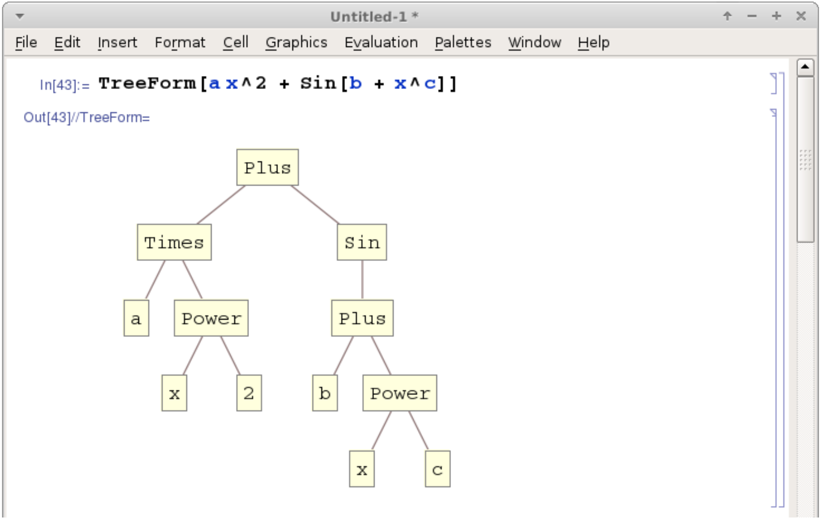
- An outline of more "idiomatic" ways to use Mathematica
- A sample of ways to use those idioms in research-like contexts
- Bi-directional!

My #1 Mathematica tip



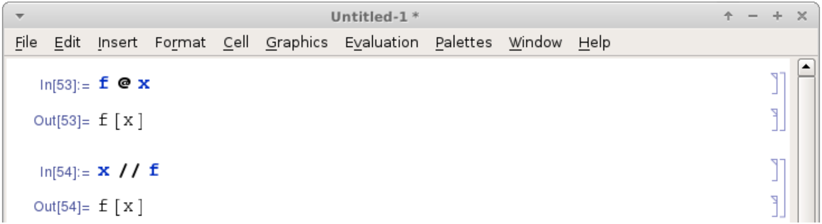
- “Reset” button for the current Mathematica session; completely removes all variables and definitions
- Sure, you could just run the `Remove["Global‘*"]` cell, but buttons are more ~~fun~~ convenient.

A little bit of syntactic sugar



A little bit of syntactic sugar

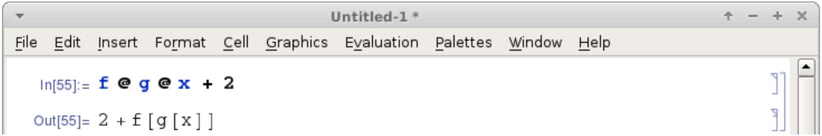
- Generally, we write math with infix notation
- Mathematica also offers prefix and postfix operators for single-argument functions:



- Cuts down on tedious bracket-matching, but beware associativity and operator precedence!

A little bit of syntactic sugar

- @ right-associates and has a high precedence:



- // left-associates and has a low precedence:

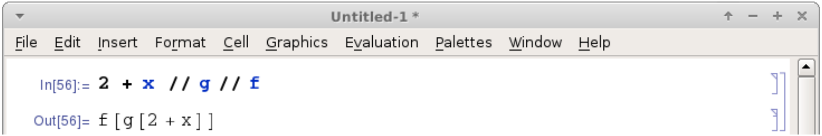


Table of Contents

"History"

1936: Alan Turing

Alan Turing invents every programming language that will ever be but is shanghaied by British Intelligence to be 007 before he can patent them.

"History"

1936: Alan Turing

Alan Turing invents every programming language that will ever be but is shanghaied by British Intelligence to be 007 before he can patent them.

1936: Alonzo Church

Alonzo Church also invents every language that will ever be but does it better. His lambda-calculus is ignored because it is insufficiently C-like. This criticism occurs in spite of the fact that C has not yet been invented.

—James Iry

What is functional programming?

- Programs as functions from inputs to outputs
- Higher-order functions
    - Functions become a sort of datatype
- Avoids mutability/state (!!!!)
- Mathematical by construction (category theory, formal computation)
- "What things *are* vs. what things *do*."
- Lots of list manipulation

Pure functions

- No side-effects: functions depend only on inputs

```
f = Function[x, x + 3]
```

Pure functions

- No side-effects: functions depend only on inputs

```
f = Function[x, x + 3]
```

- Alternatively,

```
g = # + 3&;
```

Pure functions

- No side-effects: functions depend only on inputs

```
f = Function[x, x + 3]
```

- Alternatively,

```
g = # + 3&;
```

- Multiple arguments:

```
 In[1] := h = #1 + 2*#2&;
          h[3, 4]
Out[1] := 11
```

Pure functions

- No side-effects: functions depend only on inputs

```
f = Function[x, x + 3]
```

- Alternatively,

```
g = # + 3&;
```

- Multiple arguments:

```
 In[1] := h = #1 + 2*#2&;
          h[3, 4]
Out[1] := 11
```

- Use `Block`, `With`, or `Module` to localize variables in more complicated function structures

Transforming Data

Consider applying a simple (pure!) function to a set of data. . .

- . . . naïvely, with a for-loop:

```
For[i = 1, i < Length[input], i++,
  output[[i]] = Sin[input[[i]]],
]
```

Transforming Data

Consider applying a simple (pure!) function to a set of data...

- ...naïvely, with a for-loop:

```
For[i = 1, i < Length[input], i++,
  output[[i]] = Sin[input[[i]]],
]
```

- ...with a Table command:

```
output = Table[Sin[input[[i]]], {i,1,n}]
```

(like a list comprehension in python!)

Transforming Data

Consider applying a simple (pure!) function to a set of data…

- …naïvely, with a for-loop:

```
For[i = 1, i < Length[input], i++,
  output[[i]] = Sin[input[[i]]],
]
```

- …with a Table command:

```
output = Table[Sin[input[[i]]], {i,1,n}]
```

(like a list comprehension in python!)

- …with a Map:

```
output = Map[Sin, input]
```

Transforming Data

Consider applying a simple (pure!) function to a set of data...

- ...naïvely, with a for-loop:

```
For[i = 1, i < Length[input], i++,
  output[[i]] = Sin[input[[i]]],
]
```

- ...with a Table command:

```
output = Table[Sin[input[[i]]], {i,1,n}]
```

(like a list comprehension in python!)

- ...with a Map:

```
output = Map[Sin, input]
```

- ...by cheating with the Listable attribute:

```
output = Sin[input]
```

Higher-order Functions: Map

Map applies a function to each element of a collection without modifying the original.

```
 In[1] := Map[f,{1,2,3,x,y,z}]
Out[1] := {f[1],f[2],f[3],f[x],f[y],f[z]}
```

- Automatically handles length
- Easily parallelized with ParallelMap
- Common enough to warrant special syntax:

```
 In[2] := f/@{1,2,3,x,y,z}
Out[2] := {f[1],f[2],f[3],f[x],f[y],f[z]}
```

Higher-order functions: Apply

Apply turns a list of things into formal arguments of a function—it essentially "strips off" a set of {}.

- Similar to Map, transforms a list:

```
 In[1] := Apply[f, {1, 2, 3, a, b, c}]
Out[1] := f[1, 2, 3, a, b, c]
```

- Can operate on levels[1] (default = 0, use @@@ for level 1)

```
 In[2] := Apply[f, {{1},{2},{3}}, {1}]
Out[2] := {f[1], f[2], f[3]} (*level 1*)
```

- Plus & Subtract become really useful wtih Apply

[1] # of indices required to specify element

Higher-order functions: Nest & NestList

- `Nest` repeatedly applies a function to an expression
- `NestList` does the same, producing a list of the intermediate results

- Captures iteration as a recursive application of functions

```
 In[1] := Nest[f, x, 3]
Out[1] := f[f[f[x]]]
```

Conclusion

While `Map`, `Apply`, & `Nest` are all built-in functions, none *rely* on ideas exclusive to Mathematica; as functional constructs, they very naturally capture specific types of problems & ideas.

Table of Contents

Patterns

What is a pattern?

Patterns represent classes of expressions which can be used to "automatically" simplify or restructure expressions. For example, `f[_]` and `f[x_]` both represent the pattern of "a function named `f` with anything as its argument", but `f[x_]` gives the name `x` to the argument (whatever it is).

Common patterns:

- `x_`: anything (with "the anything" given the name `x`)
- `x_Integer`: any integer (given the name `x`)
- `x_^n_`: anything to any explicit power (guess their names)
- `f[r_,r_]`: a function with two identical arguments
- and so on

The Replacement Idiom

"/. applies a rule or list of rules in an attempt to transform each subpart of an expression"

```
 In[1] := {x, x^2, y, z} /. x -> a
Out[1] := {a, a^2, y, z}
```

- The rule can make use of Mathematica's pattern-matching capabilities:

```
 In[2] := 1 + x^2 + x^4 /. x^p_ -> f[p]
Out[2] := 1 + f[2] + f[4]
```

- Useful for structuring solvers:

```
f = x /. DSolve[x''[t] == x[t], x, t][[1]]
```

Attributes

`Attributes` let you define general properties of functions, without necessarily giving explicit values.

- The `Listable` attribute automatically threads a function over lists that appear as arguments.

```
 In[1] := SetAttributes[f, Listable]
          f[{1,2,3},x]
Out[1] := {f[1,x], f[2,x], f[3,x]}
```

- `Flat`, `Orderless` used to define things like associativity & commutativity (`a+b == b+a` for the purposes of pattern matching)

Table of Contents

Some final thoughts

1. Functional programming and pattern matching are both hard and obtuse (at first), but they can be a very elegant way of attacking problems
   - Also good for parallel programing!
2. The best method usually requires a bit of trial-and-error. Experiment!
3. Further resources:
   - The Mathematica documentation is *excellent*
   - The Wolfram Blog frequently has cool examples in a variety of subjects
   - Essential Mathematica for Students of Science has *lots* of detailed notebooks for scientific applicaitons
   - Power Programming with Mathematica: antequated, but good

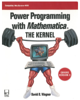


Figure: `https://www.msu.edu/~glosser1/works.html`

Thanks for listening!