# Functions in Mathematica

## Introduction to functions and Fibonacci[]

We've already seen multiple examples of functions in Mathematica like Sin[], FindRoot[] Plot[] NSolve[] Integrate[], etc. Essentially, a function is just something which takes one set of inputs and returns some output.

In order to learn about functions a bit better, I want to next introduce another built in Mathematica function and see how we can implement it on our own: Fibonacci[]

When this function is given an integer value n, it returns the nth Fibonacci number. Where the Fibonacci numbers are defined as follows:

$$F_n = F_{n-1} + F_{n-2} \text{ with } F_0 = 0 \text{ and } F_1 = 1.$$

For example the first 20 Fibonacci numbers are shown below:

| n | 0 | 1 | 2 | 3 | 4 | 5 | 6 | 7 | 8 | 9 | 10 | 11 | 12 | 13 | 14 | 15 | 16 | 17 | 18 | 19 | 20 |
|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|
| $F_n$ | 0 | 1 | 1 | 2 | 3 | 5 | 8 | 13 | 21 | 34 | 55 | 89 | 144 | 233 | 377 | 610 | 987 | 1597 | 2584 | 4181 | 6765 |

One can also show that $F_n = \frac{2^{-n}\left(-\left(1-\sqrt{5}\right)^n+\left(1+\sqrt{5}\right)^n\right)}{\sqrt{5}}$

Try for yourself:

In[ ]:=
```
Fibonacci[100]
```

Now let's learn how we could write a function that does this on our own. First we will use the explicit definition for the nth Fibonacci number

```
(*Follow along*)
```

## Spoiler

This same form can be used to write out any simple function regardless of the number of parameters. For instance, we we want a simple function which returns $x^2 + y^2$ we could write on like this:

In[ ]:=
```
f[x_, y_] := x^2 + y^2
f[3, 4]
```

Out[ ]=

25

## What if we want more complicated functions?

For those of you who have written code in other languages before, you may have noticed a big difference between the function we just declared and the ones you're used to. Usually, functions are given their own scope. For instance, in something like python we could declare a function as follows:

(Also note that depending on how or if you have python installed on your system this may or may not run. Nevertheless, python isn't the point of this class so don't sweat it.)

In[ ]:=

```
def simple_multiplication(a, b):
    ans = 0
    for i in range(b):
        ans += a
    return ans

simple_multiplication(3,5)
```

Out[ ]=

```
15
```

The main point here is that in most languages you can write multi-line functions, and if you declare a variable inside the function it doesn't affect any variables defined outside of that function. For instance, once again in python:

In[ ]:=

```
ans = 10
def simple_multiplication(a, b):
    ans = 0
    for i in range(b):
        ans += a
    return ans

print(ans)
print(simple_multiplication(3,5))
print(ans)
```

```
10
15
10
```

But how can we do this in Mathematica????

```
(*Follow along*)
```

## Spoiler

What is the difference between Module and Block?

Module gives all scoped variables their own unique symbols

In[ ]:=
```
a = 2;
Dynamic[Row[{"The current value of a is: ", a}]]

Module[{a},
 Pause[1];
 a = 10;
 Pause[1];
 Row[{"Module returns: ", Hold[a], "=", a}]
]
Row[{"The ending value of a is: ", a}]
```

While Block temporarily changes the actual values of each variable but then resets them after it is done running

In[ ]:=
```
a = 2;
Dynamic[Row[{"The current value of a is: ", a}]]
Block[{a},
 Pause[1];
 a = 10;
 Pause[1];
 Row[{"Module returns: ", Hold[a], "=", a}]
]
Row[{"The ending value of a is: ", a}]
```

**NOTE: Block can cause some really really weird behavior, so unless you know what you're doing I recommend using Module**

In fact there's actually a current bug where Dynamic doesn't even update after block exits as you can see above. I almost always use Module instead of Block.

## Exercise: Can you re-write the Fibonacci function again without using the explicit definition?

Here is how the code could work in python.

It finds the nth Fibonacci number the same way you might try to do it manually. It starts with a and b equaling 0 and 1 and then uses them to count up starting with 0+1=1 then 1+1=2 then 1+2=3 then 2+3=5, etc., until we reach the nth Fibonacci number

In[ ]:=
```
def fibonacci(n):
    if n <= 0:
        return 0
    elif n == 1:
        return 1
    else:
        a, b = 0, 1  # Initialize the first two Fibonacci numbers
        for i in range(1, n):
            a, b = b, a + b  # Update the values of a and b
        return b  # b is the nth Fibonacci number
```

```
fibonacci(10)
```

Out[ ]=

55

Now see if you can write it in Mathematica using what we learned.

To get you started I've written the code that goes in the function for you. Your job is just to turn it into a function.

In[ ]:=

```
n = 10;

(*Make this a function of n instead of requiring n to be declared above*)
If[n ≤ 0, Return 0];
If[n == 1, Return 1];
a = 0;
b = 1;
Do[
  {a, b} = {b, a + b}
  , {i, 1, n - 1}];
b
```

## Recursive functions

Another topic that is often taught in functional programing is the idea of recursive functions, and in fact this is the best way to implement a numerical algorithm for finding the Fibonacci numbers. The idea here is that our function actually calls itself from inside itself. Once again looking at python code for an example:

In[ ]:=

```
def RecursiveFibonacci(n):
    if(n==0): return 0;
    if(n==1): return 1;
    return RecursiveFibonacci(n-1) + RecursiveFibonacci(n-2)

RecursiveFibonacci(10)
```

Out[ ]=

55

We can see this results in a function which is very easy to read and implement. And we could actually do the exact same thing to implement it in Mathematica like this:

In[ ]:=
```
BadRecursiveFibonacci[n_] := Module[{},
  If[n == 0, Return[0]];
  If[n == 1, Return [1]];
  BadRecursiveFibonacci[n - 1] + BadRecursiveFibonacci[n - 2]
 ]
BadRecursiveFibonacci[10]
```

Out[ ]=
```
55
```

But this is where I want to make a very important point. **Often times writing a function the same way you would in a typical language is not the best way to do it in Mathematica**. In fact, there is actually an even easier and better way to write a recursive function in Mathematica which I'll show below:

In[ ]:=
```
GoodRecursiveFibonacci[0] = 0;
GoodRecursiveFibonacci[1] = 1;
GoodRecursiveFibonacci[n_] :=
 GoodRecursiveFibonacci[n - 1] + GoodRecursiveFibonacci[n - 2]
GoodRecursiveFibonacci[10]
```

Out[ ]=
```
55
```

Mathematica has the really cool ability to give specific values of functions their own definitions. So instead of having to write cases into our function we could simply write out the function the exact same way you would write out the definition of the Fibonacci numbers. In fact, the whole idea of functions in Mathematica is sort of a lie. Whenever Mathematica sees a definition with a pattern in it (i.e, n_ in this case) it tries to replace anything that matches the pattern with the definition. However, hard-coded definitions take priority so we can use this to implement cases. We'll learn more about patterns in Mathematica next week, but one such pattern we could use is n_Integer instead of just n_. This will only allow the function to simplify for integer values of n.

In[ ]:=
```
BetterRecursiveFibonacci[0] = 0;
BetterRecursiveFibonacci[1] = 1;
BetterRecursiveFibonacci[n_Integer] :=
 BetterRecursiveFibonacci[n - 1] + BetterRecursiveFibonacci[n - 2]
BetterRecursiveFibonacci[10]

BetterRecursiveFibonacci[10.5]
GoodRecursiveFibonacci[10.5]
```

Out[ ]=
```
55
```

Out[ ]=
```
BetterRecursiveFibonacci[10.5]
```

$RecursionLimit: Recursion depth of 1024 exceeded.

Out[ ]=
```
TerminatedEvaluation[RecursionLimit] + GoodRecursiveFibonacci[10.5 - 2]
```

We can see that the BetterRecursiveFibonacci function called with n=10.5 doesn't get simplified, but GoodRecursiveFibonacci tries to plug 10.5 into the recursive definition which causes the base cases of n=0 and n=10 to never be hit, which eventually leads to an error caused by the recursion never terminating.

## Even better recursive functions

These recursive functions are really nice as we've seen, but we can actually make them even better using one more very useful trick in Mathematica which can allow for some really *really* efficient code.

Right now, our BeterRecursiveFibonacci function works pretty well for small numbers but let's look at how slow it is with big numbers

In[•]:=
```
BetterRecursiveFibonacci[0] = 0;
BetterRecursiveFibonacci[1] = 1;
BetterRecursiveFibonacci[n_Integer] :=
 BadRecursiveFibonacci[n - 1] + BadRecursiveFibonacci[n - 2]
AbsoluteTiming[BetterRecursiveFibonacci[30]]
```

Out[•]=
```
{5.72964, 832 040}
```

Over 5 whole seconds to compute the 30th Fibonacci!?!?! Awful! But why is it so bad?

Let's make a new function which prints out each time it is called so we can get a better idea of what's going on.

In[•]:=
```
TrackRecursiveFibonacci[0] = 0;
TrackRecursiveFibonacci[1] = 1;
TrackRecursiveFibonacci[n_] := Module[{},
  Print["Called with n = ", n];
  TrackRecursiveFibonacci[n - 1] + TrackRecursiveFibonacci[n - 2]
 ]

TrackRecursiveFibonacci[8]
```

```
Called with n = 8
Called with n = 7
Called with n = 6
Called with n = 5
Called with n = 4
Called with n = 3
Called with n = 2
Called with n = 2
Called with n = 3
Called with n = 2
Called with n = 4
Called with n = 3
Called with n = 2
Called with n = 2
Called with n = 5
Called with n = 4
Called with n = 3
Called with n = 2
Called with n = 2
Called with n = 3
Called with n = 2
Called with n = 6
Called with n = 5
Called with n = 4
Called with n = 3
Called with n = 2
Called with n = 2
Called with n = 3
Called with n = 2
Called with n = 4
Called with n = 3
Called with n = 2
Called with n = 2
```

Out[ ]=

```
21
```

Look at all that repetition! We run the same computations over and over again. Wouldn't it be much better if there was a way we could tell Mathematica to simply remember previous evaluations instead

of computing them from scratch each time?

Well thankfully there is. And in fact whenever you have a slow function that you need to call multiple times this can be very helpful.

In[•]:=
```
Clear[BestRecursiveFibonacci]
BestRecursiveFibonacci[0] = 0;
BestRecursiveFibonacci[1] = 1;
BestRecursiveFibonacci[n_Integer] :=
  BestRecursiveFibonacci[n] = BestRecursiveFibonacci[n - 1] + BestRecursiveFibonacci[n - 2];

AbsoluteTiming[BestRecursiveFibonacci[1000]]
```

Out[•]=
```
{0.0033943,
 43 466 557 686 937 456 435 688 527 675 040 625 802 564 660 517 371 780 402 481 729 089 536 555 417 949 ⁖
  051 890 403 879 840 079 255 169 295 922 593 080 322 634 775 209 689 623 239 873 322 471 161 642 996 440 ⁖
  906 533 187 938 298 969 649 928 516 003 704 476 137 795 166 849 228 875}
```

Look at that! We were able to compute the 1000th Fibonacci number orders of magnitude faster than we were able to compute only the 30th before! Why does this work?

Each time we call BestRecursiveFibonacci[] we run the following code
**BestRecursiveFibonacci[*n*] = BestRecursiveFibonacci[*n* – 1] + BestRecursiveFibonacci[*n* – 2]**
This causes Mathematica to hard code the value of BestRecursiveFibonacci[n] each time it is run, the same way we hard coded the values for 0 and 1. This way, Mathematica essentially remembers all previous evaluations.

Finally, if you want to make it forget these past evaluations, then you just need to use Clear[BestRecursiveFibonacci] like usual and it will clear all these hard-coded definitions.

## Some Mathematica weirdness - It's functions all the way down

Now that we should have a good understanding of functions, I want to explain another thing that is really unique to Mathematica. In Mathematica **pretty much everything is just a function!** In fact, Mathematica doesn't even have different types, it just renders different functions in different ways. For instance, when we have a list in Mathematica it is just something of the form List[a,b,c]

In[•]:=
```
Clear[a, b, c]
myList = {a, b, c}
```

Out[•]=
```
{a, b, c}
```

When we define a list like this, Mathematica is actually just displaying the code List[a,b,c] in this shorthand notation as {a,b,c}. In other words, Mathematica has some built in mappings that cause certain functions to be displayed in their own unique ways. So while we can write {a,b,c} Mathematica is essentially just pretending every instance of {a,b,c} is actually just the List[] function applied to the

sequence a,b,c.

The terminology Mathematica uses for this is saying that the “Head” of your list is the List function.

Mathematica explains this as follows: https://reference.wolfram.com/language/tutorial/Expressions.html#4715

## Everything Is an Expression

The Wolfram Language handles many different kinds of things: mathematical formulas, lists, and graphics, to name a few. Although they often look very different, the Wolfram Language represents all of these things in one uniform way. They are all *expressions*.

A prototypical example of a Wolfram Language expression is f[x, y]. You might use f[x, y] to represent a mathematical function $f(x, y)$. The function is named f, and it has two arguments, x and y.

You do not always have to write expressions in the form $f[x, y, \ldots]$. For example, x+y is also an expression. When you type in x+y, the Wolfram Language converts it to the standard form Plus[x, y]. Then, when it prints it out again, it gives it as x+y.

The same is true of other "operators", such as ^ (Power) and / (Divide).

In fact, everything you type into the Wolfram Language is treated as an expression.

| | |
|---|---|
| x+y+z | **Plus**[x, y, z] |
| xyz | **Times**[x, y, z] |
| x^n | **Power**[x, n] |
| {a, b, c} | **List**[a, b, c] |
| a->b | **Rule**[a, b] |
| a=b | **Set**[a, b] |

*Some examples of Wolfram Language expressions.*

Oftentimes it can also be very useful to understand what “expression” or function Mathematica is using to represent something in the background. These enclosing expressions are known as “Heads” in the wolfram language. And the function Head[] will map some expression to its resulting head. For instance:

In[ ]:=
```
Head[{a, b, c, d}]
Head[2]
Head[2.5]
Head[2 + 3 i]
```

Head[  ]

Out[ ]=
```
List
```

Out[ ]=
```
Integer
```

Out[ ]=
```
Real
```

Out[ ]=
```
Complex
```

Out[ ]=
```
Graphics3D
```

## Some other ways to apply functions

We already saw that we can call functions as follows:

In[ ]:=
```
Clear[f]
f[4]
```

Out[ ]=
```
f[4]
```

### @ and //

But there are actually a few other ways we can call functions first instead of typing the square brackets around our input. Two common ways are shown below:

In[ ]:=
```
f@4
4 // f
```

Out[ ]=
```
f[4]
```

Out[ ]=
```
f[4]
```

We can see that both of these forms do the same thing, and apply f to 4. So why would we use either of these notations?

Well, the whole reason we use Mathematica in the first place is to make life easy. So lets say you want to quickly apply multiple functions to the same object in succession (kinda like the Linux pipeline), then instead of wasting tons of time typing a bunch of square brackets we could just do the following:

```
In[•]:= TextRecognize@Binarize@ColorNegate@ImageSegmentationComponents
Out[•]=
ImageSegmentationComponents
```

Ok, so that can be pretty helpful in some cases, but why do we also need // ? Well this can be really nice when you want to quickly change the output of a given function. For instance lets go back to our original explicit Fibonacci function:

```
In[•]:= MyFibonacci[n_] := (2^-n (-(1 - Sqrt[5])^n + (1 + Sqrt[5])^n))/Sqrt[5];

MyFibonacci[10]
Out[•]=
```

$$\frac{-\left(1-\sqrt{5}\right)^{10}+\left(1+\sqrt{5}\right)^{10}}{1024\sqrt{5}}$$

By default Mathematica displays the exact form of the answer in terms of all the roots and exponents. But what if we want a numeric version? Well then we can call the N function on the result

```
In[•]:= N[MyFibonacci[10]]
Out[•]=
55.
```

However a lot of the time it makes more sense to do this “type” conversion at the end of the line so you could type something like this instead. It’s much quicker to type, and more readable to some extent.

```
In[•]:= MyFibonacci[10] // N
Out[•]=
55.
```

Another example I use all the time is in displaying matrices. For instance if we want to row reduce a matrix we could do the following:

```
In[•]:= RowReduce[{{1, 2, 3, 1}, {5, 4, 2, 2}, {5, -1, 3 I, 5}}]
Out[•]=
```

$$\left\{\left\{1,\ 0,\ 0,\ \frac{2332}{3133}-\frac{792\,i}{3133}\right\},\ \left\{0,\ 1,\ 0,\ -\frac{171}{241}+\frac{99\,i}{241}\right\},\ \left\{0,\ 0,\ 1,\ \frac{1749}{3133}-\frac{594\,i}{3133}\right\}\right\}$$

But the resulting answer isn’t that readable cause it’s in list form. So we can call MatrixForm on it to make it easier to read using //MatrixForm (go try adding this to the end of the line above yourself)

## @@, /@ and @@@

Ok ok, what schizophrenic bullshit is that subsubsection title? Well these are a few other ways you can

apply functions but with some special meaning.

@@ is actually very simple, it simply means “Replace Head.” For instance if we have g@@f[x] it will replace the “f” head with the “g” head as follows:

```
In[•]:= g @@ f[x]
Out[•]=
g[x]
```

This is sort of like type casting in other languages. And it can be especially useful if you want to map a list into a function for instance.

```
In[•]:= g @@ {a, b, c, d}
Out[•]=
g[a, b, c, d]
```

Next /@ simply means Map a function over a list. For instance:

```
In[•]:= f /@ {a, b, c, d}
Out[•]=
{f[a], f[b], f[c], f[d]}
```

We can see it applies f to every element in the list

Finally @@@ is like a combination of /@ and @@, which replaces the head of every element in a list. For instance

```
In[•]:= g @@@ {f[a], f[b], f[c]}
Out[•]=
{g[a], g[b], g[c]}
```

I rarely ever use this one tbh, but I use @@ and /@ all the time. Especially /@

## Pure functions

Finally, I want to talk about pure functions. These are the equivalent to lambda functions in python if you’re familiar with those, or more generally anonymous functions in other languages.

In Mathematica they can be defined in two ways. The first is using Function[]

```
In[•]:= Function[{x, y}, x^2 + y^2][a, b]
Out[•]=
a^2 + b^2
```

This can be useful when you want to do something in a single line instead of declaring it separately. For instance let’s say we wanted to raise every item in a list to the power of itself. We could do this in one line as follows:

```
In[•]:= Function[{x}, x^x] /@ {a, b, c, d}
Out[•]=
{a^a, b^b, c^c, d^d}
```

Finally, Mathematica also has some shorthand notation for dealing with these pure functions using the & symbol where

**Function[{x1,x2,x3,x4...},*f*[x1,x2,x3,x4,...]] ⟺ *f*[#1,#2,#3,#4,...]&**

Using this, we can do the same $x^x$ example again as follows:

```
In[•]:= #^# & /@ {a, b, c, d}
```

Out[•]=

$$\{a^a, b^b, c^c, d^d\}$$

Or something like this as another example

```
In[•]:= (#1^2 + #2^2) &[a, b]
```

Out[•]=

$$a^2 + b^2$$

## Final Exercise

See if you can predict the output of these 3 code cells before you evaluate the cell. If you can, you're already an expert with Mathematica notation!

If you get stuck try writing some of your own code to evaluate parts of it instead of just checking the answer right away!

```
In[•]:= Head /@ {1, 1.1, 2 + 3 i}
```

```
In[•]:= #1^#2 & @@@ {{x1, y1}, {x2, y2}, {x3, y3}}
```

```
In[•]:= Clear[h];
h[0] = 1;
h[1] = 2;
h[a_] := h[a - 2];

h@102 / h @@ p[101] // N
```