# 1 Functions

A mathematical function is defined abstractly by

> **Function:** A function is a *relation* which assigns to each element in the domain a *single* element in the range. A **relation** is a set of ordered pairs, $(x, y)$. The set of first coordinates is the domain, the set of second coordinates the range of the relation.

That is, a function gives a correspondence between values in its domain with values in its range.

This definition is abstract, as functions can be very general. With single-variable calculus, we generally specialize to real-valued functions of a single variable (*univariate functions*). These typically have the correspondence given by a rule, such as $f(x) = x^2$ or $f(x) = \sqrt{x}$. The function's domain may be implicit (as in all $x$ for which the rule is defined) or may be explicitly given as part of the rule. The function's range is then the image of its domain, or the set of all $f(x)$ for each $x$ in the domain ($\{f(x) : x \in \text{ domain}\}$).

Some examples of mathematical functions are:

$$f(x) = \cos(x), \quad g(x) = x^2 - x, \quad h(x) = \sqrt{x}, \quad s(x) = \begin{cases} -1 & x < 0 \\ 1 & x > 0 \end{cases}.$$

For these examples, the domain of both $f(x)$ and $g(x)$ is all real values of $x$, where as for $h(x)$ it is implicitly just the set of non-negative numbers, $[0, \infty)$. Finally, for $s(x)$, we can see that the domain is defined for every $x$ but 0.

In general the range is harder to identify than the domain, and this is the case for these functions too. For $f(x)$ we know the cos function is trapped in $[-1, 1]$ and it is intuitively clear than all values in that set are possible. The function $h(x)$ would have range $[0, \infty)$. The $s(x)$ function is either $-1$ or 1, so only has two possible values in its range. What about $g(x)$? It is a parabola that opens upward, so any $y$ values below the $y$ value of its vertex will not appear in the range. In this case, the symmetry indicates that the vertex will be at $(1/2, -1/4)$, so the range is $[-1/4, \infty)$.

**Thanks to Euler (1707-1783):** The formal idea of a function is a relatively modern concept in mathematics. According to Dunham, Euler defined a function as an "analytic expression composed in any way whatsoever of the variable quantity and numbers or constant quantities." He goes on to indicate that as Euler matured, so did his notion of function, ending up closer to the modern idea of a correspondence not necessarily tied to a particular formula or "analytic expression." He finishes by saying: "It is fair to say that we now study functions in analysis because of him."

We will see that defining functions within `Julia` can be as simple a concept as Euler started with, but that the more abstract concept has a great advantage that is exploited in the design of the language.

## 1.1 Defining simple mathematical functions

The notation `Julia` uses to define simple mathematical functions could not be more closely related to how they are written mathematically. For example, the functions $f(x)$, $g(x)$, and $h(x)$ above may be defined by:

```
f(x) = cos(x)
g(x) = x^2 - x
h(x) = sqrt(x)
```

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

The left-hand sign of the equals sign is still an assignment, though in this case an assignment to a function object which has a name and a specification of an argument, $x$ in each case above, though other *dummy variables* could be used. The right hand side is simply `Julia` code to compute the *rule* corresponding to the function.

Calling the function also follows standard math notation:

```
f(pi), g(2), h(4)
```

```
(-1.0, 2, 2.0)
```

For typical cases like the three above, there isn't really much new to learn.

> The equals sign in the definition of a function above is an *assignment*. Assignment restricts the expressions available on the *left*-hand side to a) a variable name, b) an indexing assignment, as in `xs[1]`, or c) a function assignment following this form `function_name(args...)`. Whereas function definitions and usage in `Julia` mirrors standard math notation; equations in math are not so mirrored in `Julia`. In mathematical equations, the left-hand of an equation is typically a complicated algebraic expression. Not so with `Julia`, where the left hand side of the equals sign is prescribed and quite limited.

### 1.1.1 The domain of a function

Functions in `Julia` have an implicit domain, just as they do mathematically. In the case of $f(x)$ and $g(x)$, the right-hand side is defined for all real values of $x$, so the domain is all $x$. For $h(x)$ this isn't the case, of course. Trying to call $h(x)$ when $x < 0$ will give an error:

```
h(-1)
```

```
Error: DomainError with -1.0:
sqrt will only return a complex result if called with a complex argument. T
ry sqrt(Complex(x)).
```

The `DomainError` is one of many different error types `Julia` has, in this case it is quite apt: the value $-1$ is not in the domain of the function.

### 1.1.2 Equations, functions, calling a function

Mathematically we tend to blur the distinction between the equation

$$y = 5/9 \cdot (x - 32)$$

and the function

$$f(x) = 5/9 \cdot (x - 32)$$

In fact, the graph of a function $f(x)$ is simply defined as the graph of the equation $y = f(x)$. There is a distinction in `Julia` as a command such as

```
x = -40
y = 5/9 * (x - 32)
```

- 4 0 . 0

will evaluate the righthand side with the value of `x` bound at the time of assignment to `y`, whereas assignment to a function

```
f(x) = 5/9 * (x - 32)
f(72)                                   ## room temperature
```

2 2 . 2 2 2 2 2 2 2 2 2 2 2 2 2 2

will create a function object which is called with a value of `x` at a later time - the time the function is called. So the value of `x` defined when the function is created is not important here (as the value of `x` used by `f` is passed in as an argument).

Within `Julia`, we make note of the distinction between a function object versus a function call. In the definition `f(x)=cos(x)`, the variable `f` refers to a function object, whereas the expression `f(pi)` is a function call. This mirrors the math notation where an $f$ is used when properties of a function are being emphasized (such as $f \circ g$ for composition) and $f(x)$ is used when the values related to the function are being emphasized (such as saying "the plot of the equation $y = f(x)$).

Distinguishing these three related but different concepts (equations, function objects, and function calls) is important when modeling on the computer.

### 1.1.3 Cases

The definition of $s(x)$ above has two cases:

$$s(x) = \begin{cases} -1 & s < 0 \\ 1 & s > 0. \end{cases}$$

We learn to read this as "If $s$ is less than 0, then the answer is $-1$. If $s$ is greater than 0 the answer is 1." Often - but not in this example - there is an "otherwise" case to catch those values of $x$ that are not explicitly mentioned. As there is no such "otherwise" case here, we can see that this function has no definition when $x = 0$. This function is often called the "sign" function and is also defined by $|x|/x$. (`Julia`'s `sign` function actually defines `sign(0)` to be `0`.)

How do we create conditional statements in `Julia`? Programming languages generally have "if-then-else" constructs to handle conditional evaluation. In `Julia`, the following code will handle the above condition:

```
if x < 0
  -1
elseif x > 0
   1
end
```

The "otherwise" case would be caught with an `else` addition. So, for example, this would implement `Julia`'s definition of `sign` (which also assigns 0 to 0):

```
if x < 0
  -1
elseif x > 0
   1
else
   0
end
```

The conditions for the `if` statements are expressions that evaluate to either `true` or `false`, such as generated by the Boolean operators `<`, `<=`, `==`, `!-`, `>=`, and `>`.

If familiar with `if` conditions, they are natural to use. However, for simpler cases of "if-else" `Julia` provides the more convenient *ternary* operator: `cond ? if_true : if_false`. (The name comes from the fact that there are three arguments specified.) The ternary operator checks the condition and if true returns the first expression, whereas if the condition is false the second condition is returned. Both expressions are evaluated. (The short-circuit operators can be used to avoid both evaluations.)

For example, here is one way to define an absolute value function:

```
f(x) = x >= 0 ? x : -x
```

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

The condition is `x >= 0` - or is `x` non-negative? If so, the value `x` is used, otherwise `-x` is used.

Here is a means to implement a function which takes the larger of `x` or `10`:

```
f(x) = x > 10 ? x : 10.0
```

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

(This could also utilize the `max` function: `f(x) = max(x, 10.0)`.)

Or similarly, a function to represent a cell phone plan where the first 500 minutes are 20 dollars and every additional minute is 5 cents:

```
cellplan(x) = x < 500 ? 20.0 : 20.0 + 0.05 * (x-500)
```

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

> Type stability. These last two definitions used `10.0` and `20.0` instead of the integers `10` and `20` for the answer. Why the extra typing? When `Julia` can predict the type of the output from the type of inputs, it can be more efficient. So when possible, we help out and ensure the output is always the same type.

**Example** The `ternary` operator can be used to define an explicit domain. For example, a falling body might have height given by $h(t) = 10 - 16t^2$. This model only applies for non-negative $t$ and non-negative $h$ values. So, in particular $0 \leq t \leq \sqrt{10/16}$. To implement this function we might have:

```
h(t) = 0 <= t <= sqrt(10/16) ? 10.0 - 16t^2 : error("t is not in the domain")
```

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

We might also have used `NaN` instead of an error, or 0, as the falling body would come to rest when it hits the ground.

**Nesting ternary operators** The function `s(x)` isn't quite so easy to implement, as there isn't an "otherwise" case. We could use an `if` statement, but instead illustrate using a second, nested ternary operator:

```
s(x) = x < 0 ? -1 : (x > 0 ? 1 : error("0 is not in the domain"))
```

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

With nested ternary operators, the advantage over the `if` condition is not very compelling, but for simple cases the ternary operator is quite useful. (The extra parentheses around the expression when `x<0` is not true are actually unnecessary, though added here for clarity.)

## 1.2 Functions defined with the "function" keyword

For more complicated functions, say one with a few steps to compute, an alternate form for defining a function can be used:

```
function function_name(function_arguments)
  ...function_body...
end
```

The last value computed is returned unless the `function_body` contains an explicit `return` statement.

For example, the following is a more verbose way to define $f(x) = x^2$:

```
function f(x)
  return x^2
end
```

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

The line `return x^2`, could have just been `x^2` as it is the last (and) only line evaluated.

> The `return` keyword is not a function, so is not called with parentheses. An emtpy `return` statement will return a value of `nothing`.

**Example** Imagine we have the following complicated function related to the trajectory of a projectile with wind resistance:

$$f(x) = \left(\frac{g}{kv_0\cos(\theta)} + \tan(\theta)\right)x + \frac{g}{k^2}\ln\left(1 - \frac{k}{v_0\cos(\theta)}x\right)$$

Here $g$ is the gravitational constant 9.8 and $v_0$, $\theta$ and $k$ parameters, which we take to be 200, 45 degrees and 1/2 respectively. With these values, the above function can be computed when $x = 100$ with:

```
function f(x)
  g, v0, theta, k = 9.8, 200, 45, 1/2
  a = v0 * cosd(theta)

  (g/(k*a) + tand(theta))* x + (g/k^2) * log(1 - k/a*x)
end
f(100)
```

9 6 . 7 5 7 7 1 7 9 1 6 3 2 1 6 1

By using a multi-line function our work is much easier to look over for errors.

## 1.3 Parameters, function context (scope), keyword arguments

Consider two functions implementing the slope-intercept form and point-slope form of a line:

$$f(x) = m \cdot x + b, \quad g(x) = y_0 + m \cdot (x - x_0).$$

Both functions use the variable $x$, but there is no confusion, as we learn that this is just a dummy variable to be substituted for and so could have any name. Both also share a variable $m$ for a slope. Where does that value come from? In practice, there is a context that gives an answer. Despite the same name, there is no expectation that the slope will be the same for each function if the context is different. So when parameters are involved, a function involves a rule and a context to give specific values to the parameters.

Something similar is also true with `Julia`. Consider the example of writing a function to model a linear equation with slope $m = 2$ and $y$-intercept 3. A typical means to do this would be to define constants, and then use the familiar formula:

```
m, b = 2, 3
f(x) = m*x + b
```

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

This will work as expected. For example, $f(0)$ will be $b$ and $f(2)$ will be 7:

```
f(0), f(2)
```

```
(3, 7)
```

All fine, but what if somewhere later the values for $m$ and $b$ were *redefined*:

```
m, b = 3, 2
```

```
(3, 2)
```

Now what happens with $f(0)$? When $f$ was defined `b` was 3, but now if we were to call `f`, `b` is 2. Which value will we get? More generally, when `f` is being evaluated in what context does `Julia` look up the bindings for the variables it encounters? It could be that the values are assigned when the function is defined, or it could be that the values for the parameters are resolved when the function is called. If the latter, what context will be used?

Before discussing this, let's just see in this case:

```
f(0)
```

2

So the `b` is found from the currently stored value. This fact can be exploited. we can write template-like functions, such as `f(x)=mx+b` and reuse them just by updating the parameters separately.

How `Julia` resolves what a variable refers to is described in detail in the manual page Scope of Variables. In this case, the function definition finds variables in the context of where the function was defined, the main workspace. As seen, this context can be modified after the function definition and prior to the function call. It is only when `b` is needed, that the context is consulted, so the most recent binding is retrieved. Contexts (more formally known as environments) allow the user to repurpose variable names without there being name collision. For example, we typically use `x` as a function argument, and different contexts allow this `x` to refer to different values.

Mostly this works as expected, but at times it can be complicated to reason about. In our example, definitions of the parameters can be forgotten, or the same variable name may have been used for some other purpose. The potential issue is with the parameters, the value for `x` is straightforward, as it is passed into the function. However, we can also pass the parameters, such as $m$ and $b$, as arguments. For parameters, we suggest using keyword arguments. These allow the specification of parameters, but also give a default value. This can make usage explicit, yet still convenient. For example, here is an alternate way of defining a line with parameters `m` and `b`:

```
f(x; m=1, b=0) = m*x + b
```

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

The right-hand side is identical to before, but the left hand side is different. Arguments defined *after* a semicolon are keyword arguments. They are specified as `var=value` (or `var::Type=value` to restrict the type) where the value is used as the default, should a value not be specified when the function is called.

Calling a function with keyword arguments can be identical to before:

```
f(0)
```

0

During this call, values for `m` and `b` are found from how the function is called, not the main workspace. In this case, nothing is specified so the defaults of $m = 1$ and $b = 0$ are used. Whereas, this call will use the user-specified values for `m` and `b`:

```
f(0, m=3, b=2)
```

2

Keywords are used to mark the parameters whose values are to be changed from the default. Though one can use *positional arguments* for parameters - and there are good reasons to do so - using keyword arguments is a good practice if performance isn't paramount, as their usage is more explicit yet the defaults mean that a minimum amount of typing needs to be done.

**Example** In the example for multi-line functions we hard coded many variables inside the body of the function. In practice it can be better to pass these in as parameters along the lines of:

```
function f(x; g = 9.8, v0 = 200, theta = 45, k = 1/2)
  a = v0 * cosd(theta)
  (g/(k*a) + tand(theta))* x + (g/k^2) * log(1 - k/a*x)
end
```

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

## 1.4 Multiple dispatch

The concept of a function is of much more general use than its restriction to mathematical functions of single real variable. A natural application comes from describing basic properties of geometric objects. The following function definitions likely will cause no great concern when skimmed over:

```
Area(w, h) = w * h                                  # of a rectangle
Volume(r, h) = pi * r^2 * h                         # of a cylinder
SurfaceArea(r, h) = pi * r * (r + sqrt(h^2 + r^2)) # of a right circular cone, including
the base
```

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

The right-hand sides may or may not be familiar, but it should be reasonable to believe that if push came to shove, the formulas could be looked up. However, the left-hand sides are subtly different - they have two arguments, not one. In `Julia` it is trivial to define functions with multiple arguments - we just did.

Earlier we saw the `log` function can use a second argument to express the base. This function is basically defined by `log(b,x)=log(x)/log(b)`. The `log(x)` value is the natural log, and this definition just uses the change-of-base formula for logarithms.

But not so fast, on the left side is a function with two arguments and on the right side the functions have one argument - yet they share the same name. How does `Julia` know which to use? `Julia` uses the number, order, and *type* of the arguments passed to a function to determine which function definition to use. This is technically known as multiple dispatch or **polymorphism**. As a feature of the language, it can be used to greatly simplify the number of functions the user must learn. The basic idea is that many functions are "generic" in that they will work for many different scenarios.

Multiple dispatch is very common in mathematics. For example, we learn different ways to add: integers (fingers, carrying), real numbers (align the decimal points), rational numbers (common denominators), complex numbers (add components), vectors (add