Functions

Initialization

```
ClearAll["Global`*"];
Off[General::spell, General::spell1]
```

Introduction

What is a function?

In the most general sense, a function is any conceptual object that accepts input and produces corresponding output based upon that input.

In this notebook we introduce many of the most important concepts and techniques used to create *Mathematica* functions;

Implicit functions

An *implicit function* is a named expression which contains one or more symbols that are considered variables. For example, below we define *f* to be a quadratic function of *x*, where *a*, *b*, and *c* are considered parameters, although here the distinction between variables and parameters is merely a matter of attitude.

```
Clear[x, a, b, c];
f = a x^2 + b x + c;
```

As presently defined, *f* is an implicit function which simply returns its symbolic definition.

```
f
```

$c + b\,x + a\,x^2$

Numerical values can be obtained either by using replacement rules to specify the values of the variables and parameters

```
f /. {a → 1, b → 2, c → 3, x → 4}
```

```
27
```

which does not alter the definition of the function

```
f
```

$c + b\,x + a\,x^2$

or by assigning values to the parameters

```
a = 1; b = 2; c = 3;
f
```

$3 + 2\,x + x^2$

and variables

```
x = 4;
f
```

```
27
```

Assignment of values precludes many of the operations we might wish to perform on such a function.

```
D[f, x]
```

```
General::ivar : 4 is not a valid variable. ≫
```

$\partial_4 27$

because *x* is now a number instead of a variable.

```
?x
```

```
Global`x

x = 4
```

Although we can undo this unfortunate assignment

```
Clear[x];
D[f, x]
```

```
2 + 2 x
```

you will soon find that unintended or obsolete assignments can be a major source of frustration to new users of *Mathematica*.

For example, if we neglect to clear the variables used on the rhs and forget that *a* and/or *b* have already been assigned values, a new function

```
g = a Sin[x] + b Cos[x]
```

```
2 Cos[x] + Sin[x]
```

does not contain the parameters *a* and *b* as intended.

If we want to have a general form without assignment in f and with assignment in g

```
Clear[a, b, c, x]
D[f, x]
D[g, x]
```

```
b + 2 a x
```

```
Cos[x] - 2 Sin[x]
```

A good technique for assigning parameters to a function is to give a set of replacement rules a name that clearly identifies it purpose and its association with the relevant function. For example, the coefficients of the quadratic function called **f** above are identified below as its parameters. Note that we employ a string variable to avoid evaluation of the expression for **f**.

```
Clear[a, b, c, x]
```

```
parameters["f"] = {a → 1, b → 2, c → 3};
```

```
f /. parameters["f"]
```

$3 + 2\,x + x^2$

```
f /. x → 2 /. parameters["f"]
```

```
11
```

This technique enhances the readability of notebooks and reduces the chance of errors due to spurious assignments.

▼ Express $f = \frac{1}{1+x^3}$ as an implicit function of x. Plot f for $y \to \cos(x)$.

▼ Express $f = \frac{1}{1+x^3}$ as an implicit function of x. Evaluate $\frac{\partial f}{\partial x}$ for $y \to \mathrm{xos}(x)$.

Explicit functions

Explicit functions receive their variables as arguments. The arguments can be expressed as symbols, patterns, or a mixture of types. The functions may be represented in terms of several rules, with the rule that applies to a particular set of arguments chosen by pattern matching.

Patterns as arguments

Often the most convenient method for specifying the arguments of an explicit function is to use patterns rather than explicit symbols. Consider a function

```
Clear[g]
```

```
g[x_] = x^3 Exp[-x^2 * Sin[x]];
```

whose argument is expressed in terms of the pattern **x_**. The single underscore is the symbol for a blank pattern that may assume any value, symbolic or numeric. Note that this usage of the symbol **x** does not conflict with any global definitions that might already exist. Thus,

```
g[x]
```

$e^{-x^2 \mathrm{Sin}[x]}\, x^3$

gives the expected result while

```
g[y]
```

$e^{-y^2 \mathrm{Sin}[y]}\, y^3$

now gives the corresponding expression with **x** replaced by **y**. Similarly,

```
g[3]
```

$27\, e^{-9 \mathrm{Sin}[3]}$

now evaluates to an exact numerical expression and

```
g[3.]
g[3] // N
```

```
7.58185
```

```
7.58185
```

evaluates to an approximate number. There is also nothing to stop us from using a string as the argument,

```
g["string"]
```

$$\text{string}^3\, e^{-\text{string}^2\, \text{Sin}[\text{string}]}$$

```
g["string"] // FullForm
```

```
Times[Power["string", 3],
 Power[E, Times[-1, Power["string", 2], Sin["string"]]]]
```

even if the result is uninterpretable as a mathematical function. It is also possible to use a function as the argument.

```
g[g[z]]
```

$$e^{-3 z^2 \text{Sin}[z] - e^{-2 z^2 \text{Sin}[z]} z^6 \text{Sin}\left[e^{-z^2 \text{Sin}[z]} z^3\right]}\, z^9$$

This technique can be very useful for recursive functions, such as iterated function systems, or for symbolic manipulation and will be explored at greater depth later.

Similarly, functions of several variables can be constructed using different symbols to identify each pattern.

$$g[x\_, y\_] = x^{3+y}\, \text{Exp}\left[-x^2 * \text{Sin}[y]\right];$$

```
{g[x, y], g[x, 0], g[x, 2], g[x, -4], g[2, 2], g[x, "string"]}
```

$$\left\{e^{-x^2\,\text{Sin}[y]}\,x^{3+y},\ x^3,\ e^{-x^2\,\text{Sin}[2]}\,x^5,\ \frac{e^{x^2\,\text{Sin}[4]}}{x},\ 32\,e^{-4\,\text{Sin}[2]},\ e^{-x^2\,\text{Sin}[\text{string}]}\,x^{3+\text{string}}\right\}$$

Each of these forms evaluates in the expected manner. However, why didn't we use a different symbol for this new function? Will a second definition for the same symbol, namely **g**, conflict with the first?

```
g[w]
```

$$e^{-w^2\,\text{Sin}[w]}\,w^3$$

There is no conflict between these definitions!

Mathematica can stores the rules associated with each symbol and then, when evaluating expressions, scans its rule tables for any rules associated with the symbols that appear in that expression, beginning with those defined by the user and proceeding to the built-in rules that apply to any symbol of the same type. The appropriate rule is selected by pattern matching. The rules associated with the symbol **g** can be inspected using the **?** operator; sometimes **??** will give even more information.

```
?g
```

```
Global`g
```

$$g[x\_] = e^{-x^2\,\text{Sin}[x]}\,x^3$$

$$g[x\_,\ y\_] = e^{-x^2\,\text{Sin}[y]}\,x^{3+y}$$

Here we find two rules associated with the symbol **g**. If our expression matches the pattern **g[x_]** the rule for that pattern is applied, or if our expression matches **g[x_,y_]** the corresponding rule is applied. There is no conflict because the patterns are unique. On the other hand, no evaluation is performed when the expression involves an unrecognized pattern.

```
g[x , y, w]
```

Therefore, this technique can be used to define generalized functions and in the design of functions with multiple definitions patterns.

▼ Explain the behavior of the following functions: g[x_]=π or h[_]=π. Are they equivalent?

▼ Explain the behavior of the following function:

```
Clear[f];
```
f[x_] = x^3;

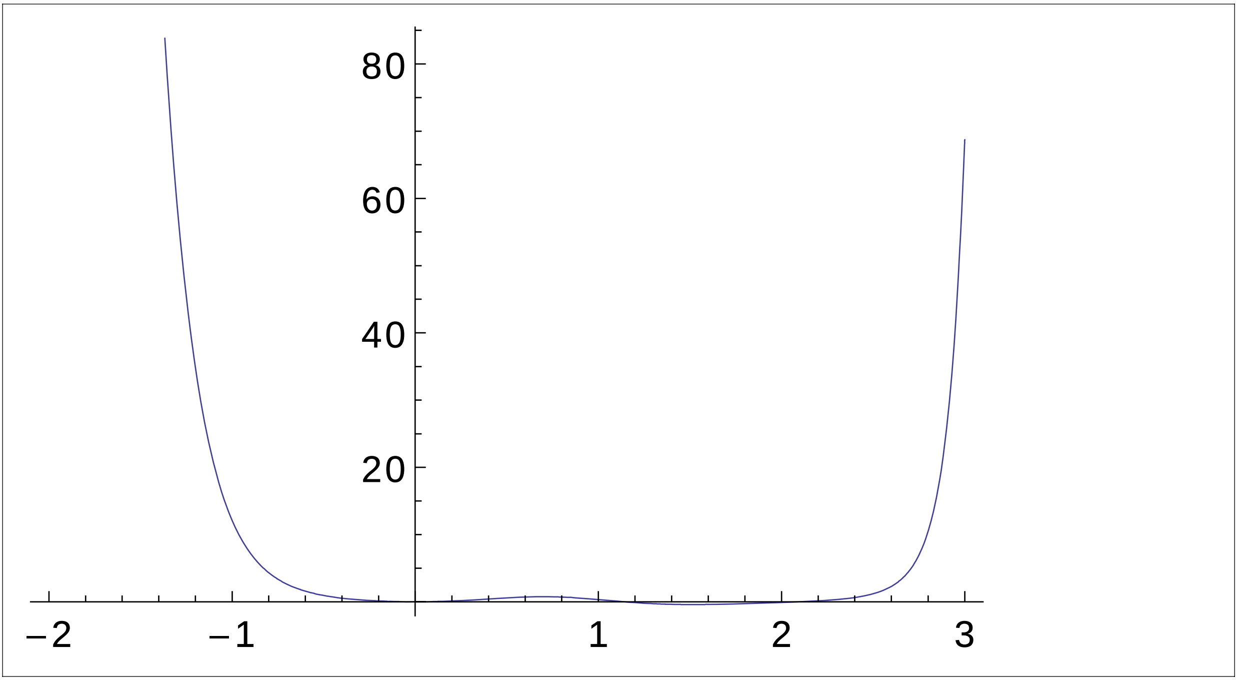
f[x_,y_] = x^{y+1};

Evaluate f[3], f[1,2], f[x,$\frac{y}{\text{Log}[x]}$]. How do these several definitions coexist peacefully?

Delayed assignment

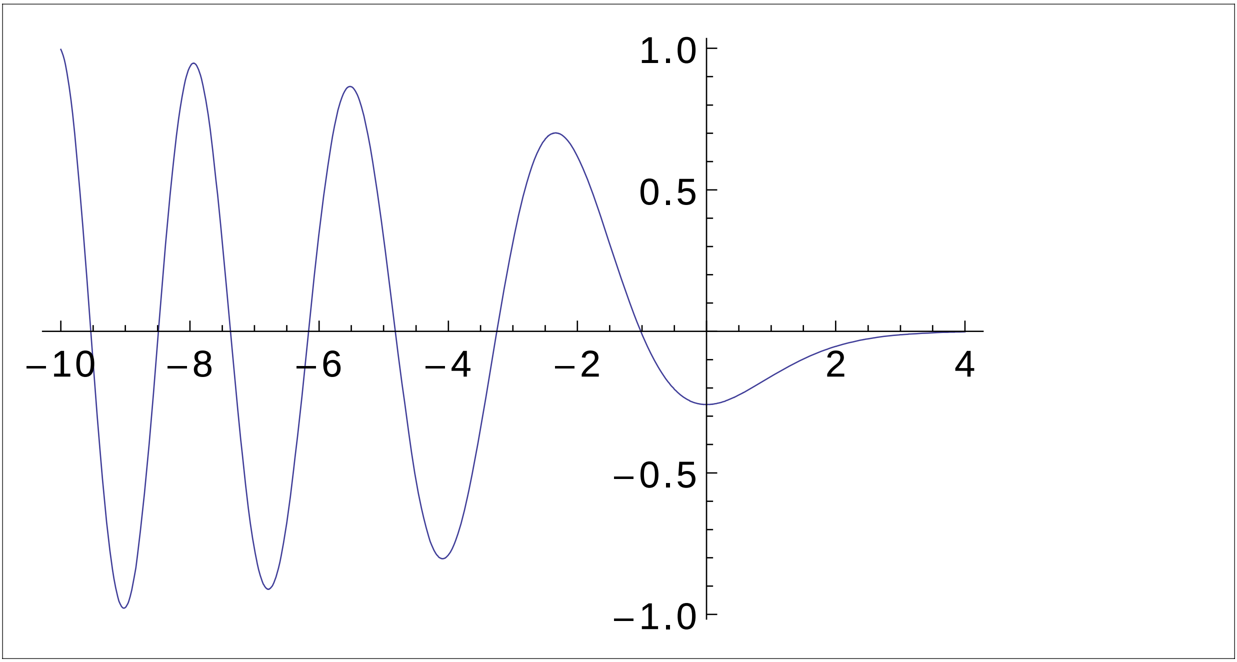
Immediate assignment in the form **`lhs=rhs`** is appropriate when **`rhs`** provides an immediate result that can be delivered without further processing of the **`lhs`**. However, if **`rhs`** is more complicated, such as the result of a plotting command, that cannot be evaluated directly, then we should use delayed assignment of the form **`lhs:=rhs`**. Consider the following function intended to produce a graph of the derivative of a function of a single variable over a specified domain.

```
PlotDerivative[g_, {x_, xmin_, xmax_}] :=
  Plot[Evaluate[D[g[x], x]], {x, xmin, xmax}]
PlotDerivative[g, {x, -2, 3}]
```



An example of the use of this function is given below.

```
PlotDerivative[AiryAi, {x, -10, 4}]
```

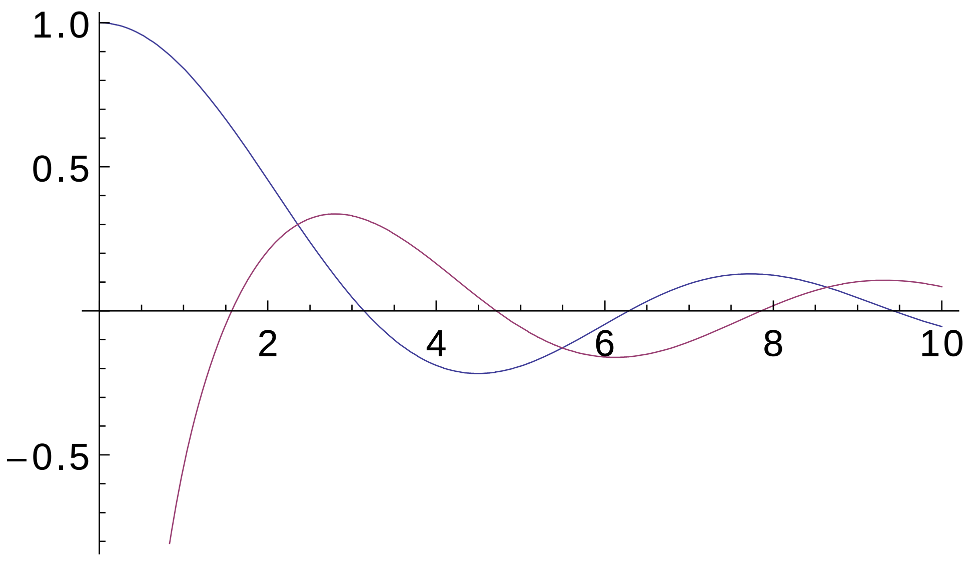


```
j1[n_, z_] := Sqrt[π/2] z^(-1/2) BesselJ[n + 1/2, z];
y1[n_, z_] := Sqrt[π/2] z^(-1/2) BesselY[n + 1/2, z];
```

$$j1[n\_, z\_] := \sqrt{\frac{\pi}{2}}\, z^{-1/2}\, \text{BesselJ}\left[n + \frac{1}{2}, z\right];$$

$$y1[n\_, z\_] := \sqrt{\frac{\pi}{2}}\, z^{-1/2}\, \text{BesselY}\left[n + \frac{1}{2}, z\right];$$

```
plotbessgen[n_, xmin_, xmax_] :=
    Plot[{j1[n, x], y1[n, x]}, {x, xmin, xmax}];
plotbessgen[0, 0.001, 10]
```



The advantage of delayed assignment is that the risk of conflict with pre-existing symbols is reduced.

- ▼ The Stirling approximation for Log[n!] is
  $\mathrm{Log}[n!] \approx n\,\mathrm{Log}[n] - n + \frac{1}{2}\mathrm{Log}[2\pi n] + \frac{1}{12n} + \cdots$.
  a) Write functions logStirling1 using just the first two terms (the lowest reasonable approximation), logStirling2 including the next term, and logStirling3 including the next two terms.
  b) Also write functions which evaluate the percentage error in these approximations.
  c) Make a LogPlot which displays the percentage errors for all three approximations as functions of n. [Hint: you will probably need to load the package Graphics`Graphics` to use LogPlot.]

- ▼ Write a new version of the Stirling function for which logStirling[n,1] gives the first approximation, logStirling[n,2] the second, and logStirling[n,3] the third. Furthermore, logStirling[n] with only a single argument should give the most accurate formula, namely logStirling[n,3].

Functions formed from command sequences

We will often need functions which require many steps to produce their results. Perhaps the simplest method for assembling a multistep function is to define a function in terms of an expression which contains a sequence of commands. The basic syntax would be:

```
f[args_] := (expr1; expr2; … ; output)
```

where the initial steps return their output to the next step rather than to the user, with only the final step returning a printed result because it does not end with a terminal semicolon. For example, the following function finds the coefficient of x^n for $(a+b\,x)^m$ using *Mathematica* functions **Expand** and **Coefficient**.

```
Clear[f, n, m]
```

```
f[n_, m_] := (pol = Expand[(a + b x)^m]; Coefficient[temp, x, n])
```

```
f[3, 13]
```

```
0
```

Although this function achieves its goal, it has the undesirable effect of leaving behind detritus in the form of unwanted symbols which should have been temporary.

```
pol
```

$$a^{13} + 13\, a^{12}\, b\, x + 78\, a^{11}\, b^2\, x^2 + 286\, a^{10}\, b^3\, x^3 + 715\, a^9\, b^4\, x^4 + 1287\, a^8\, b^5\, x^5 + 1716\, a^7\, b^6\, x^6 + 1716\, a^6\, b^7\, x^7 + 1287\, a^5\, b^8\, x^8 + 715\, a^4\, b^9\, x^9 + 286\, a^3\, b^{10}\, x^{10} + 78\, a^2\, b^{11}\, x^{11} + 13\, a\, b^{12}\, x^{12} + b^{13}\, x^{13}$$

The simple example suggested above can be written more efficiently using pure functions which do not require creation of intermediate symbols, but it is often difficult to formulate more interesting multistep functions in readable fashion without creating local variables. Traditional programming languages solve this problem by providing constructions known as subprograms to isolate the local variables which are used for intermediate steps and then are discarded when no longer needed. These subprograms also protect their local variables from corruption by external commands.

Pure functions

A pure function is represented by the basic syntax **Function[var,body]** where **var** represents the argument and **body** the definition of the function. The argument can be a single variable or a list of variables. The function is then applied to an argument using the syntax Function[var,body][arg] to replace **var** by **arg** in **body**.

Pure functions with names

If we intend to use a function several times we should assign it a name and refer to it by name rather than repeat its definition each time it is used.

```
h = Function[x, a x^2 + b x + c]
```

```
Function[x, a x^2 + b x + c]
```

```
h[Sin[ϕ]^2]
```

```
c + b Sin[ϕ]^2 + a Sin[ϕ]^4
```

```
h[h[y]]
```

```
c + b (c + b y + a y^2) + a (c + b y + a y^2)^2
```

Here **h** is simply a symbol that represents a pure function.

```
?h
```

```
Global`h

h = Function[x, a x^2 + b x + c]
```

The formal arguments used to define the body of the function are local variables that will not interfere with assignments made elsewhere for symbols with the same names. In the example below, the assignment given to **x** does not affect the definition of a function that uses **x** as its local variable.

```
x = "something";
g = Function[x, x^2];
```

```
g[y]
```

```
y^2
```

However, parameters that appear in the body of the functions are treated as global symbols. Therefore, assignments of those parameters will affect subsequent evaluation of the functions and must be made with care.

```
h[a]
```

$a^3 + a\, b + c$

```
a = 2;
h[z]
```

$c + b\, z + 2\, z^2$

```
Clear[a]
```

It is also important to recognize that functions defined in this manner are not evaluated until arguments are presented. Therefore, it is not necessary to use delayed assignment to prevent premature evaluation of the right-hand side. Notice how in the cell below the definition using **Function** behaves properly while similar definitions using immediate assignment fail, the first by failing to protect the local variable and the second by premature evaluation.

```
Clear[f1, f2, g, y, z];
f1[x_] = Expand[x];
f2[y_] = Expand[y];
g = Function[x, Expand[x]];
```

```
f1[(y + z)^2]
```

```
something
```

```
f2[(y + z)^2]
```

```
(y + z)^2
```

```
g[(y + z)^2]
```

```
y^2 + 2 y z + z^2
```

We call functions of this type *pure* because we define the behavior of the function, not its arguments. In fact, we can manipulate a pure function by name without supplying any arguments at all. For example, suppose that we require the derivative of the quadratic function defined above. The derivative can be evaluated in an obvious fashion using the built-in function **D**

```
D[h[y], y]
```

```
b + 2 a y
```

but this method requires that **h** be given an explicit argument. We prefer the pure form using the conventional prime notation

```
h'
```

```
Function[x, b + 2 a x]
```

to obtain the derivative as a pure function also. A pure function can be evaluated with any suitable argument.

```
{h'[stuff], h'[2/3]}
```

```
{b + 2 a stuff, 4a/3 + b}
```

▼ The logistic map is defined by the function f[x_]:=μ x (1-x) where μ is the growth parameter. Write a pure function that embodies this mapping. Use NestList to produce an iterated function sequence {y,f[y],f[f[y]],f[f[f[y]]],...} nested 4 times.

▼ Pure and explicit function definitions can be mixed. Study the behavior of the following function:
f[μ_]=Function[x,μ x (1-x)]
Evaluate the following expressions and interpret the results:
a) f[a] b) f[a][x] c)f[2.5][0.8] d) NestList[f[μ],y,4]

▼ Once again define:
f[μ_]=Function[x,μ x (1-x)]
Then evaluate the following expressions and interpret the results:
a) f[a] ' b) f[a]'[z] c) f[3.5]'[0.25]

Pure functions of several variables

Pure functions with several variables are defined by using a list of variable names in place of **var** and the same variable names in **body**. When the function is evaluated it must be given a corresponding set of arguments, which are substituted in order.

```
s = Function[{x, y, z}, x y^-3 Erf[z]]
```

```
Function[{x, y, z}, x Erf[z] / y^3]
```

```
Clear[x];
s[x, 1/y, 4]
```

```
x y^3 Erf[4]
```

```
s[x, x, x]
```

```
Erf[x] / x^2
```

Of course, you do need to supply enough input to obtain a complete evaluation.

```
s[1]
```

Function::fpct :

Too many parameters in {x, y, z} to be filled from Function$\left[\{x, y, z\}, \frac{x\,\mathrm{Erf}[z]}{y^3}\right]$[1]. ≫

Function$\left[\{x, y, z\}, \frac{x\,\mathrm{Erf}[z]}{y^3}\right]$[1]

Notice that even though the variables appear in **Function** without blank patterns (underscore) attached, there is no conflict between symbols used outside of the function definition and those used inside, even when the same names are involved.

```
x = "stuff"; y = "more stuff";
h = Function[{x, y}, x + y]
```

```
Function[{x, y}, x + y]
```

```
h[x, y]
```

```
more stuff + stuff
```

```
Clear[a, b];
h[a, b]
```

```
a + b
```

The positions of the arguments matter, not their names.

Clear[x, y, z];
Function$\left[\{x, y, z\}, x\,y^2\,z^3\right]$[z, x, y]

$x^2\,y^3\,z$

▼ Express as pure functions:
a) $f[x\_, y\_] := x^y$ b) g[x_,y_]:={Cos[x],Sin[y]}

Anonymous functions

Why bother giving a name to a specialized function that we will probably use just once and then discard? Omission of a name avoids storing unnecessary definitions and conserves computational resources; it also relieves us of the responsibility for conceiving a memorable name that does not conflict with other symbols. Nameless functions are obviously called *anonymous*.

```
Clear[a, b, c]
```

```
Function[x, a x^2 + b x + c]
```

```
Function[x, a x^2 + b x + c]
```

```
Function[x, a x^2 + b x + c][q]
```

```
c + b q + a q^2
```

```
Function[x, a x^2 + b x + c][Tanh[y]]
```

```
c + b Tanh[y] + a Tanh[y]^2
```

```
Function[x, a x^2 + b x + c]@Tanh[y]
```

```
c + b Tanh[y] + a Tanh[y]^2
```

Skeletal notation

Since the arguments of a pure function are only local variables, why bother naming them? If a pure function is only to be used once, why bother naming it or using the word **Function**? There is a very convenient notation that can be used to define pure functions without cluttering the world with useless names. Thus, **body&** is equivalent to **Function[var,body]** with each variable replaced by a **Slot**. If only one variable is needed, it can be represented by the unnumbered slot #; but if several are needed their slots are numbered sequentially as **#1**, **#2**, etc. MacDonald coined the term *skeletal notation* to describe this method of expressing functional relationships in bare form without fleshy names.

```
Clear[a, b, c, x]
```

```
a #^2 + b # + c &[x]
```

```
c + b x + a x^2
```

```
h = a #^2 + b # + c &
```

```
a #1^2 + b #1 + c &
```

```
h[stuff]
```

```
c + b stuff + a stuff^2
```

```
g = (#1 + 2 #2) Exp[-#3^2] &;
```

```
g[x, y, z]
```

```
ⅇ^(-z^2) (x + 2 y)
```

$e^{-z^2}\,(x+2\,y)$

Many built-in functions are pure — we can manipulate such functions without specifying arguments. For example, here we define **f1** to be the tangent function and **f2** to be its derivative.

```
Clear[f1, f2];
f1 = Tan;
f2 = f1'
```

```
Sec[#1]^2 &
```

We find that **f2** is returned as a pure function given with skeletal notation. There is no need to name the slot. If we want a particular value, we simply fill the slot with the desired argument.

```
{f1[30 Degree], f2[30 Degree]}
```

$\left\{\frac{1}{\sqrt{3}},\ \frac{4}{3}\right\}$

A pure function can be applied to a list which contains the required arguments using the **Apply** operator @@. The syntax **f@@expr** replaces the head of **expr** with **f**; thus, if the expression is a list of the form **expr=List[a,b,⋯]**, normally printed as **{a,b,⋯}**, the result is simply **f[a,b,⋯]**.

```
args = {x, y, z};
g @@ args
```

$e^{-z^2}\,(x+2\,y)$

```
FullForm[args]
```

```
List[x, y, z]
```

If the list is long enough the slots will be filled from the beginning of the list,

```
g @@ {x, y, z, "more stuff"}
```

$e^{-z^2}(x + 2\,y)$

but if it is too short the expression cannot be evaluated fully, leaving an unfilled slot.

```
g @@ {x, y}
```

Function::slotn : Slot number 3 in $(\#1 + 2\,\#2)\, e^{-\#3^2}$ & cannot be filled from $\left((\#1 + 2\,\#2)\, e^{-\#3^2}\ \&\right)$[x, y]. ≫

$e^{-\#3^2}(x + 2\,y)$

- ▼ Translate the following expressions into skeletal form and give examples of their use.
  a) $f[x\_] := x^2 \text{Cos}[x]$ b) g[x_]:=Expand[x] c) h[x_,y_,z_]:=Min[x,y,z] Max[x,y,z]

- ▼ Express the logistic map μ x (1-x) using a pattern argument for the parameter μ and a slot for the independent variable x. Then evaluate Nest[f[α],y,4] where f is your function.

- ▼ Write a skeletal function which selects the n^{th} element of a list. Test by mapping your function onto a list of the form {{a,b,c},{1,2,3},{"eeny", "meeny", "miny", "moe"}}.

- ▼ Write and test a skeletal function which produces polar coordinates {r,θ} from Cartesian coordinates {x,y}.

Example: simplification using skeletal pure functions

Often one develops a symbolic result in a series of steps, with the result of each step fed to a function which performs the next transformation. Such calculations are conveniently performed using the postfix operator (//) to add each step of the sequence as the calculation is developed.

For example, suppose we wish to create a function which produces the coefficient of x^3 that appears in the expansion of $(a + b\,x)^{13}$. A simple way to accomplish this task is to supply the symbolic expansion to a skeletal function that extracts the desired coefficient. The result of the first step, the expansion, is the argument that fills the slot in the next step reserved by #. The presence of & indicates an anonymous pure function.

```
Expand[(a + b x)^13] // Coefficient[#, x, 3] &
```

```
286 a^10 b^3
```

Generalizing this procedure, we can write a function with extracts the coefficient of x^n from the expansion of $(a + b\,x)^m$ for arbitrary n, m. Notice that we still use delayed assignment to prevent premature evaluation of the part of the definition using **Expand**.

```
Clear[f];
f[n_, m_] := Expand[(a + b x)^m] // Coefficient[#, x, n] &
```

```
f[3, 13]
```

```
286 a^10 b^3
```

Notice that the internal representation employs the standard form **(body)&[arg]**, although I prefer to define such functions using postfix form.

```
?f
```

```
Global`f
```

```
f[n_, m_] := (Coefficient[#1, x, n] &)[Expand[(a + b x)^m]]
```

An important virtue of this style of coding is that it avoids creation of unwanted symbols for intermediate results. Therefore, we often use anonymous pure functions to perform multistep transformations in a single cell without creating superfluous definitions for intermediate results.

Example: interpolating functions

Many *Mathematica* functions, such as **Interpolation** or **NDSolve**, return results in the form of interpolating functions. An **InterpolatingFunction** is used to represent an approximate function whose values are obtained by interpolation within a table. Thus, **InterpolatingFunction[domain,table]** is a pure function such that **InterpolatingFunction[domain,table][x]** computes the function at argument **x** defined within a specified **domain** using values contained in a **table**. In standard output only the domain is printed, with the table indicated schematically by <>. Note that if the argument is outside the domain, a warning message is issued and an extrapolated value is returned; however, extrapolation tends to go bad very quickly outside the domain of the interpolating function and should be avoided.

The motion of a pendulum is described by the differential equation $\theta''[t] == -\text{Sin}[\theta[t]]$. The following function evaluates that motion for a range of times assuming initial conditions $\theta[0] == \theta 0$ and $\theta'[0] == \omega 0$. Notice that both the differential equation and the initial conditions must be formulated as equations, using **==** instead of **=**. It is also necessary to specify the time range. We use delayed assignment and give arguments in the form of blank patterns.

```
swing[θ0_, ω0_, tmin_, tmax_] :=
 NDSolve[{θ''[t] == -Sin[θ[t]], θ[0] == θ0, θ'[0] == ω0}, θ, {t, tmin, tmax}]
```

A particular evaluation for specific times and initial conditions

```
swing[3π/4, 0, 0, 10]
```

```
{{θ → InterpolatingFunction[{{0., 10.}}, <>]}}
```

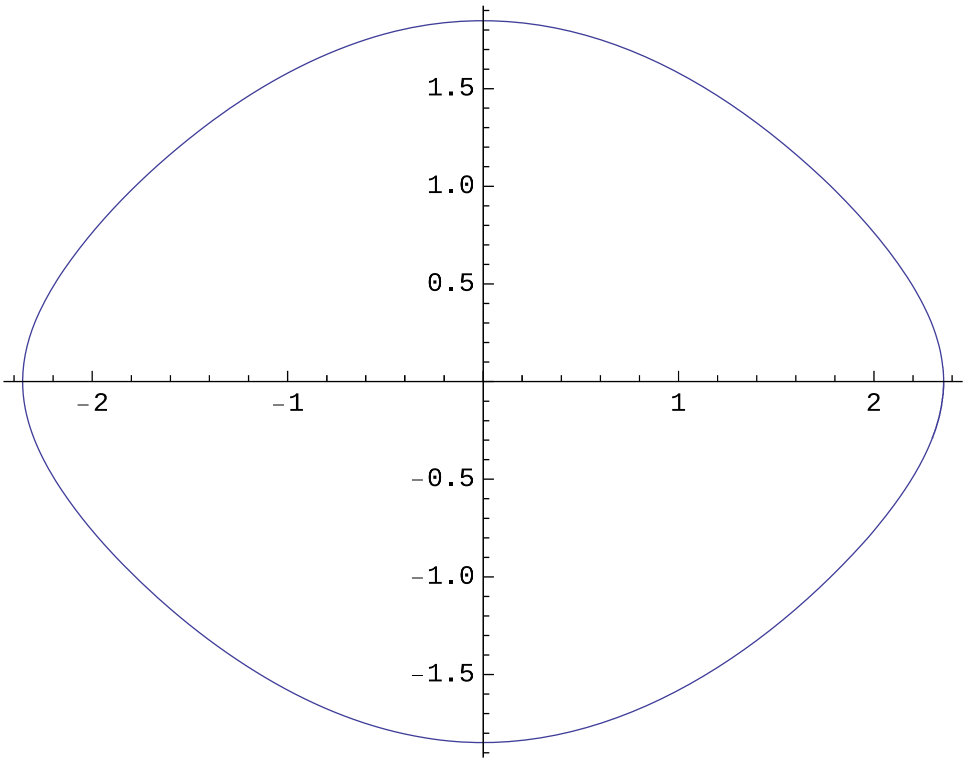
returns a solution in the form of a replacement rule which replaces θ by an interpolating function. We can now produce an explicit function of **t** by using this replacement rule.

```
swing1[t] = θ[t] /. swing[3π/4, 0, 0, 10][[1]]
```

```
InterpolatingFunction[{{0., 10.}}, <>][t]
```

A phase diagram of angular velocity versus angle can then be obtained using **ParametricPlot**.

```
ParametricPlot[Evaluate[{swing1[t], ∂t swing1[t]}],
 {t, 0, 10}, AspectRatio → Automatic]
```



Notice that the phase trajectory is flattened, like an American football, whereas the phase trajectory for simple harmonic motion would be circular. The pendulum approaches the simple harmonic oscillator as the amplitude of the swing approaches zero.

▼ Produce a figure that displays phase trajectories for initial angles of $\left\{\theta_0, \frac{\pi}{10}, 9\frac{\pi}{10}, \frac{\pi}{10}\right\}$, where we use *Mathematica* iterator notation. [Hint: Table can be used to form a list of pairs of functions of the form $\{\{\theta_1[t],\ \theta_1'[t]\}, \{\theta_2[t],\ \theta_2'[t]\} \cdots\}$, where each pair represents the phase trajectory for a different initial condition, and ParametricPlot can plot this entire family of trajectories simultaneously.]

Recursive functions

A function is described as recursive if it is defined in terms of itself. Perhaps the simplest example is the factorial function defined by the recursion relation factorial$[n]$ = n factorial$[n-1]$ with initial condition factorial$[0] = 1$.

```
factorial[0] = 1;
factorial[n_] := n factorial[n - 1]
```

Unfortunately, this function as written behaves poorly if given an argument which is not a nonnegative integer. We will learn how to restrict such functions shortly, but at the present time we prefer to focus on the operation of valid recursive functions without worrying about such problems.

```
Table[factorial[n], {n, 0, 10}]
```

```
{1, 1, 2, 6, 24, 120, 720, 5040, 40320, 362880, 3628800}
```

The simple function given above is somewhat inefficient because each time it is evaluated it must assemble an entire sequence of values. This can be a serious inefficiency for functions based upon a complicated recursion relation.

```
Trace[factorial[10]]
```

```
{factorial[10], 10 factorial[10 - 1],
 {{10 - 1, 9}, factorial[9], 9 factorial[9 - 1], {{9 - 1, 8}, factorial[8],
   8 factorial[8 - 1], {{8 - 1, 7}, factorial[7], 7 factorial[7 - 1],
    {{7 - 1, 6}, factorial[6], 6 factorial[6 - 1], {{6 - 1, 5}, factorial[5],
      5 factorial[5 - 1], {{5 - 1, 4}, factorial[4], 4 factorial[4 - 1],
       {{4 - 1, 3}, factorial[3], 3 factorial[3 - 1], {{3 - 1, 2}, factorial[2],
         2 factorial[2 - 1], {{2 - 1, 1}, factorial[1], 1 factorial[1 - 1],
          {{1 - 1, 0}, factorial[0], 1}, 1 × 1, 1}, 2 × 1, 2}, 3 × 2, 6},
       4 × 6, 24}, 5 × 24, 120}, 6 × 120, 720}, 7 × 720, 5040},
   8 × 5040, 40 320}, 9 × 40 320, 362 880}, 10 × 362 880, 3 628 800}
```

A more efficient technique, called *dynamic programming*, is to use a definition which remembers its values.

```
factorial2[0] = 1;
factorial2[n_] := factorial2[n] = n factorial2[n - 1]
```

The first call for a dynamic recursion relation may actually slow the evaluation because rules must be established for the values it will remember later, but subsequent evaluations will be simpler and faster.

```
factorial[100] // Timing
```

```
{0.,
 93 326 215 443 944 152 681 699 238 856 266 700 490 715 968 264 381 621 468 592 963 \
  895 217 599 993 229 915 608 941 463 976 156 518 286 253 697 920 827 223 758 251 185 \
  210 916 864 000 000 000 000 000 000 000 000}
```

```
factorial2[100] // Timing
```

```
{0.,
 93 326 215 443 944 152 681 699 238 856 266 700 490 715 968 264 381 621 468 592 963 \
  895 217 599 993 229 915 608 941 463 976 156 518 286 253 697 920 827 223 758 251 185 \
  210 916 864 000 000 000 000 000 000 000 000}
```

```
factorial2[100] // Timing
```

```
{0.,
 93 326 215 443 944 152 681 699 238 856 266 700 490 715 968 264 381 621 468 592 963 ⋱
  895 217 599 993 229 915 608 941 463 976 156 518 286 253 697 920 827 223 758 251 185 ⋱
  210 916 864 000 000 000 000 000 000 000 000}
```

```
Trace[factorial2[10]]
```

```
{factorial2[10], 3 628 800}
```

However, dynamic programing costs memory to store the rules. Compare the rules associated with **`factorial`** versus **`factorial2`**.

```
? factorial
```

```
Global`factorial

factorial[0] = 1

factorial[n_] := n factorial[n - 1]
```

```
? factorial2
```

```
Global`factorial2

factorial2[0] = 1

factorial2[1] = 1

factorial2[2] = 2

factorial2[3] = 6

factorial2[4] = 24

factorial2[5] = 120

factorial2[6] = 720

factorial2[7] = 5040

factorial2[8] = 40 320

factorial2[9] = 362 880
```

```
factorial2[10] = 3 628 800

factorial2[11] = 39 916 800

factorial2[12] = 479 001 600

factorial2[13] = 6 227 020 800

factorial2[14] = 87 178 291 200

factorial2[15] = 1 307 674 368 000

factorial2[16] = 20 922 789 888 000

factorial2[17] = 355 687 428 096 000

factorial2[18] = 6 402 373 705 728 000

factorial2[19] = 121 645 100 408 832 000

factorial2[20] = 2 432 902 008 176 640 000

factorial2[21] = 51 090 942 171 709 440 000

factorial2[22] = 1 124 000 727 777 607 680 000

factorial2[23] = 25 852 016 738 884 976 640 000

factorial2[24] = 620 448 401 733 239 439 360 000

factorial2[25] = 15 511 210 043 330 985 984 000 000

factorial2[26] = 403 291 461 126 605 635 584 000 000

factorial2[27] = 10 888 869 450 418 352 160 768 000 000

factorial2[28] = 304 888 344 611 713 860 501 504 000 000

factorial2[29] = 8 841 761 993 739 701 954 543 616 000 000

factorial2[30] = 265 252 859 812 191 058 636 308 480 000 000

factorial2[31] = 8 222 838 654 177 922 817 725 562 880 000 000

factorial2[32] = 263 130 836 933 693 530 167 218 012 160 000 000

factorial2[33] = 8 683 317 618 811 886 495 518 194 401 280 000 000

factorial2[34] = 295 232 799 039 604 140 847 618 609 643 520 000 000

factorial2[35] = 10 333 147 966 386 144 929 666 651 337 523 200 000 000
```

```
factorial2[36] = 371 993 326 789 901 217 467 999 448 150 835 200 000 000

factorial2[37] = 13 763 753 091 226 345 046 315 979 581 580 902 400 000 000

factorial2[38] = 523 022 617 466 601 111 760 007 224 100 074 291 200 000 000

factorial2[39] = 20 397 882 081 197 443 358 640 281 739 902 897 356 800 000 000

factorial2[40] = 815 915 283 247 897 734 345 611 269 596 115 894 272 000 000 000

factorial2[41] = 33 452 526 613 163 807 108 170 062 053 440 751 665 152 000 000 000

factorial2[42] =
 1 405 006 117 752 879 898 543 142 606 244 511 569 936 384 000 000 000

factorial2[43] =
 60 415 263 063 373 835 637 355 132 068 513 997 507 264 512 000 000 000

factorial2[44] =
 2 658 271 574 788 448 768 043 625 811 014 615 890 319 638 528 000 000 000

factorial2[45] =
 119 622 220 865 480 194 561 963 161 495 657 715 064 383 733 760 000 000 000

factorial2[46] =
 5 502 622 159 812 088 949 850 305 428 800 254 892 961 651 752 960 000 000 000

factorial2[47] =
 258 623 241 511 168 180 642 964 355 153 611 979 969 197 632 389 120 000 000 000

factorial2[48] =
 12 413 915 592 536 072 670 862 289 047 373 375 038 521 486 354 677 760 000 000 000

factorial2[49] =
 608 281 864 034 267 560 872 252 163 321 295 376 887 552 831 379 210 240 000 000 000

factorial2[50] =
 30 414 093 201 713 378 043 612 608 166 064 768 844 377 641 568 960 512 000 000 000 ⋱
  000

factorial2[51] =
 1 551 118 753 287 382 280 224 243 016 469 303 211 063 259 720 016 986 112 000 000 ⋱
  000 000

factorial2[52] =
 80 658 175 170 943 878 571 660 636 856 403 766 975 289 505 440 883 277 824 000 000 ⋱
  000 000
```

```
factorial2[53] =
 4 274 883 284 060 025 564 298 013 753 389 399 649 690 343 788 366 813 724 672 000 ⋱
  000 000 000

factorial2[54] =
 230 843 697 339 241 380 472 092 742 683 027 581 083 278 564 571 807 941 132 288 ⋱
  000 000 000 000

factorial2[55] =
 12 696 403 353 658 275 925 965 100 847 566 516 959 580 321 051 449 436 762 275 840 ⋱
  000 000 000 000

factorial2[56] =
 710 998 587 804 863 451 854 045 647 463 724 949 736 497 978 881 168 458 687 447 ⋱
  040 000 000 000 000

factorial2[57] =
 40 526 919 504 877 216 755 680 601 905 432 322 134 980 384 796 226 602 145 184 481 ⋱
  280 000 000 000 000

factorial2[58] =
 2 350 561 331 282 878 571 829 474 910 515 074 683 828 862 318 181 142 924 420 699 ⋱
  914 240 000 000 000 000

factorial2[59] =
 138 683 118 545 689 835 737 939 019 720 389 406 345 902 876 772 687 432 540 821 ⋱
  294 940 160 000 000 000 000

factorial2[60] =
 8 320 987 112 741 390 144 276 341 183 223 364 380 754 172 606 361 245 952 449 277 ⋱
  696 409 600 000 000 000 000

factorial2[61] =
 507 580 213 877 224 798 800 856 812 176 625 227 226 004 528 988 036 003 099 405 ⋱
  939 480 985 600 000 000 000 000

factorial2[62] =
 31 469 973 260 387 937 525 653 122 354 950 764 088 012 280 797 258 232 192 163 168 ⋱
  247 821 107 200 000 000 000 000

factorial2[63] =
 1 982 608 315 404 440 064 116 146 708 361 898 137 544 773 690 227 268 628 106 279 ⋱
  599 612 729 753 600 000 000 000 000

factorial2[64] =
 126 886 932 185 884 164 103 433 389 335 161 480 802 865 516 174 545 192 198 801 ⋱
  894 375 214 704 230 400 000 000 000 000
```

```
factorial2[65] =
 8 247 650 592 082 470 666 723 170 306 785 496 252 186 258 551 345 437 492 922 123 ⋱
  134 388 955 774 976 000 000 000 000 000

factorial2[66] =
 544 344 939 077 443 064 003 729 240 247 842 752 644 293 064 388 798 874 532 860 ⋱
  126 869 671 081 148 416 000 000 000 000 000

factorial2[67] =
 36 471 110 918 188 685 288 249 859 096 605 464 427 167 635 314 049 524 593 701 628 ⋱
  500 267 962 436 943 872 000 000 000 000 000

factorial2[68] =
 2 480 035 542 436 830 599 600 990 418 569 171 581 047 399 201 355 367 672 371 710 ⋱
  738 018 221 445 712 183 296 000 000 000 000 000

factorial2[69] =
 171 122 452 428 141 311 372 468 338 881 272 839 092 270 544 893 520 369 393 648 ⋱
  040 923 257 279 754 140 647 424 000 000 000 000 000

factorial2[70] =
 11 978 571 669 969 891 796 072 783 721 689 098 736 458 938 142 546 425 857 555 362 ⋱
  864 628 009 582 789 845 319 680 000 000 000 000 000

factorial2[71] =
 850 478 588 567 862 317 521 167 644 239 926 010 288 584 608 120 796 235 886 430 ⋱
  763 388 588 680 378 079 017 697 280 000 000 000 000 000

factorial2[72] =
 61 234 458 376 886 086 861 524 070 385 274 672 740 778 091 784 697 328 983 823 014 ⋱
  963 978 384 987 221 689 274 204 160 000 000 000 000 000

factorial2[73] =
 4 470 115 461 512 684 340 891 257 138 125 051 110 076 800 700 282 905 015 819 080 ⋱
  092 370 422 104 067 183 317 016 903 680 000 000 000 000 000

factorial2[74] =
 330 788 544 151 938 641 225 953 028 221 253 782 145 683 251 820 934 971 170 611 ⋱
  926 835 411 235 700 971 565 459 250 872 320 000 000 000 000 000

factorial2[75] =
 24 809 140 811 395 398 091 946 477 116 594 033 660 926 243 886 570 122 837 795 894 ⋱
  512 655 842 677 572 867 409 443 815 424 000 000 000 000 000 000

factorial2[76] =
 1 885 494 701 666 050 254 987 932 260 861 146 558 230 394 535 379 329 335 672 487 ⋱
  982 961 844 043 495 537 923 117 729 972 224 000 000 000 000 000 000
```

```
factorial2[77] =
 145 183 092 028 285 869 634 070 784 086 308 284 983 740 379 224 208 358 846 781 \
  574 688 061 991 349 156 420 080 065 207 861 248 000 000 000 000 000 000

factorial2[78] =
 11 324 281 178 206 297 831 457 521 158 732 046 228 731 749 579 488 251 990 048 962 \
  825 668 835 325 234 200 766 245 086 213 177 344 000 000 000 000 000 000

factorial2[79] =
 894 618 213 078 297 528 685 144 171 539 831 652 069 808 216 779 571 907 213 868 \
  063 227 837 990 693 501 860 533 361 810 841 010 176 000 000 000 000 000 000

factorial2[80] =
 71 569 457 046 263 802 294 811 533 723 186 532 165 584 657 342 365 752 577 109 445 \
  058 227 039 255 480 148 842 668 944 867 280 814 080 000 000 000 000 000 000

factorial2[81] =
 5 797 126 020 747 367 985 879 734 231 578 109 105 412 357 244 731 625 958 745 865 \
  049 716 390 179 693 892 056 256 184 534 249 745 940 480 000 000 000 000 000 000

factorial2[82] =
 475 364 333 701 284 174 842 138 206 989 404 946 643 813 294 067 993 328 617 160 \
  934 076 743 994 734 899 148 613 007 131 808 479 167 119 360 000 000 000 000 000 \
  000

factorial2[83] =
 39 455 239 697 206 586 511 897 471 180 120 610 571 436 503 407 643 446 275 224 357 \
  528 369 751 562 996 629 334 879 591 940 103 770 870 906 880 000 000 000 000 000 \
  000

factorial2[84] =
 3 314 240 134 565 353 266 999 387 579 130 131 288 000 666 286 242 049 487 118 846 \
  032 383 059 131 291 716 864 129 885 722 968 716 753 156 177 920 000 000 000 000 \
  000 000

factorial2[85] =
 281 710 411 438 055 027 694 947 944 226 061 159 480 056 634 330 574 206 405 101 \
  912 752 560 026 159 795 933 451 040 286 452 340 924 018 275 123 200 000 000 000 \
  000 000 000

factorial2[86] =
 24 227 095 383 672 732 381 765 523 203 441 259 715 284 870 552 429 381 750 838 764 \
  496 720 162 249 742 450 276 789 464 634 901 319 465 571 660 595 200 000 000 000 \
  000 000 000

factorial2[87] =
 2 107 757 298 379 527 717 213 600 518 699 389 595 229 783 738 061 356 212 322 972 \
  511 214 654 115 727 593 174 080 683 423 236 414 793 504 734 471 782 400 000 000 \
  000 000 000 000
```

```
factorial2[88] =
 185 482 642 257 398 439 114 796 845 645 546 284 380 220 968 949 399 346 684 421 ⋱
  580 986 889 562 184 028 199 319 100 141 244 804 501 828 416 633 516 851 200 000 ⋱
  000 000 000 000 000

factorial2[89] =
 16 507 955 160 908 461 081 216 919 262 453 619 309 839 666 236 496 541 854 913 520 ⋱
  707 833 171 034 378 509 739 399 912 570 787 600 662 729 080 382 999 756 800 000 ⋱
  000 000 000 000 000

factorial2[90] =
 1 485 715 964 481 761 497 309 522 733 620 825 737 885 569 961 284 688 766 942 216 ⋱
  863 704 985 393 094 065 876 545 992 131 370 884 059 645 617 234 469 978 112 000 ⋱
  000 000 000 000 000 000

factorial2[91] =
 135 200 152 767 840 296 255 166 568 759 495 142 147 586 866 476 906 677 791 741 ⋱
  734 597 153 670 771 559 994 765 685 283 954 750 449 427 751 168 336 768 008 192 ⋱
  000 000 000 000 000 000 000

factorial2[92] =
 12 438 414 054 641 307 255 475 324 325 873 553 077 577 991 715 875 414 356 840 239 ⋱
  582 938 137 710 983 519 518 443 046 123 837 041 347 353 107 486 982 656 753 664 ⋱
  000 000 000 000 000 000 000

factorial2[93] =
 1 156 772 507 081 641 574 759 205 162 306 240 436 214 753 229 576 413 535 186 142 ⋱
  281 213 246 807 121 467 315 215 203 289 516 844 845 303 838 996 289 387 078 090 ⋱
  752 000 000 000 000 000 000 000

factorial2[94] =
 108 736 615 665 674 308 027 365 285 256 786 601 004 186 803 580 182 872 307 497 ⋱
  374 434 045 199 869 417 927 630 229 109 214 583 415 458 560 865 651 202 385 340 ⋱
  530 688 000 000 000 000 000 000 000

factorial2[95] =
 10 329 978 488 239 059 262 599 702 099 394 727 095 397 746 340 117 372 869 212 250 ⋱
  571 234 293 987 594 703 124 871 765 375 385 424 468 563 282 236 864 226 607 350 ⋱
  415 360 000 000 000 000 000 000 000

factorial2[96] =
 991 677 934 870 949 689 209 571 401 541 893 801 158 183 648 651 267 795 444 376 ⋱
  054 838 492 222 809 091 499 987 689 476 037 000 748 982 075 094 738 965 754 305 ⋱
  639 874 560 000 000 000 000 000 000 000

factorial2[97] =
 96 192 759 682 482 119 853 328 425 949 563 698 712 343 813 919 172 976 158 104 477 ⋱
  319 333 745 612 481 875 498 805 879 175 589 072 651 261 284 189 679 678 167 647 ⋱
  067 832 320 000 000 000 000 000 000 000
```

```
factorial2[98] =
 9 426 890 448 883 247 745 626 185 743 057 242 473 809 693 764 078 951 663 494 238 \
  777 294 707 070 023 223 798 882 976 159 207 729 119 823 605 850 588 608 460 429 \
  412 647 567 360 000 000 000 000 000 000 000

factorial2[99] =
 933 262 154 439 441 526 816 992 388 562 667 004 907 159 682 643 816 214 685 929 \
  638 952 175 999 932 299 156 089 414 639 761 565 182 862 536 979 208 272 237 582 \
  511 852 109 168 640 000 000 000 000 000 000 000

factorial2[100] =
 93 326 215 443 944 152 681 699 238 856 266 700 490 715 968 264 381 621 468 592 963 \
  895 217 599 993 229 915 608 941 463 976 156 518 286 253 697 920 827 223 758 251 \
  185 210 916 864 000 000 000 000 000 000 000 000

factorial2[n_] := factorial2[n] = n factorial2[n - 1]
```

Therefore, it is not always obvious that dynamic programming is preferred — which resource needs to be conserved, time or memory? Furthermore, there are limits to recursion by either method.

```
factorial[1000] // Timing
```

```
$RecursionLimit::reclim : Recursion depth of 256 exceeded. >>
```

```
{1.33357 × 10^-17,
 489 389 408 225 290 311 733 092 901 271 992 122 747 641 930 014 438 185 921 791 436 \
   206 203 124 861 273 025 602 658 192 090 352 943 295 471 568 899 276 859 871 968 129 \
   952 549 363 970 896 016 506 594 138 054 672 607 994 039 115 598 724 139 602 867 552 \
   752 460 206 879 430 251 446 817 528 881 717 616 934 579 431 722 318 828 917 753 927 \
   863 094 684 402 529 408 602 652 094 537 266 301 773 040 625 850 796 648 709 211 671 \
   816 414 540 017 964 294 868 447 481 393 994 640 014 498 204 474 536 640 252 807 911 \
   188 136 566 096 842 869 176 697 806 703 766 345 735 093 564 476 910 994 554 407 047 \
   378 605 292 907 974 902 286 464 572 369 344 980 579 211 841 524 720 878 301 517 143 \
   042 856 054 709 851 849 017 908 071 948 335 664 474 153 572 807 392 769 368 353 720 \
   505 251 236 285 155 885 705 425 792 224 234 438 958 210 381 693 387 131 958 993 780 \
   199 337 331 381 988 858 776 006 413 168 857 429 132 024 858 489 651 200 000 000 000 \
   000 000 000 000 000 000 000 000 000 000 000 000 000 000 000 000 000 000
  Hold[factorial[747 - 1]]}
```

Although such limits can sometimes be increased, usually it is better to rethink the problem!

```
ClearAll[factorial2]
```

Interest is compounded monthly at rate *mpr*, monthly percentage rate. Write a recursive function for the monthly balance given a unit initial investment.

▼ Write a recursive version of the double factorial function, which is defined for positive integers as dfact[n] = $n(n-2)\cdots n_{\text{min}}$ where $n_{\text{min}} = 1$ for odd or $n_{\text{min}} = 2$ for even integers. What happens if you give this function a negative or noninteger argument? The next section deals with restrictions on arguments.

▼ Fibonacci numbers satisfy the recursion relation $F_n = F_{n-1} + F_{n-2}$ with $F_1 = F_2 = 1$. Write a recursive function for F_n using dynamic programming. Note that you can make your definition for $F_{\text{n\_}}$. [Do not worry about illegal arguments at this time.]

Conditions on arguments

Type checking

A function definition can be restricted to arguments of a particular type using the syntax **x_type** where **type** may be **Integer**, **Real**, **Complex**, **List**, or **Symbol**.

```
Clear[f];
f[x_Integer] := x^2;
f[x_Real] := x^3;
f[x_Complex] := Abs[x]^2;
f[x_List] := (Plus @@ x)^2
```

This peculiar function gives different results for arguments that have the same numerical values but different types.

```
{f[2], f[2.]}
```

```
{4, 8.}
```

```
f[1 + I]
```

```
2
```

Here **Plus@@x** sums a list, which is then squared to produce the desired result.

```
f[{1, 2, 3, 4}]
```

```
100
```

However, **f[a]** remains unevaluated because **a** is a symbol of unknown type.

```
f[a]
```

```
f[a]
```

```
f[x_Symbol] := "whatever you want"
```

```
f[a]
```

```
whatever you want
```

This still doesn't give a result for $f[a^2]$ because a^2 does not match the pattern **_Symbol**.

```
f[a^2]
```

```
f[a^2]
```

When in doubt, use the function **MatchQ** to determine whether an expression matches a pattern.

```
{MatchQ[a, _Symbol], MatchQ[a^2, _Symbol]}
```

```
{True, False}
```

```
?f
```

```
Global`f
```

```
f[x_Integer] := x^2

f[x_Real] := x^3

f[x_Complex] := Abs[x]^2

f[x_List] := (Plus @@ x)^2

f[x_Symbol] := whatever you want
```

▼ Write a recursive factorial function that only responds to integer input.

Definitions with conditions

Conditions can be applied to the rules that define an explicit function. This technique can be used with multiple definitions to produce functions whose limiting cases evaluate properly or to define discontinuous or piecewise functions. The basic syntax uses the **Condition** operator (**/;**) to apply conditions to one or more pattern arguments

```
f[⋯, y_ /; test, z_, ⋯ ] := rhs
```

or to a definition

```
f[x_] := rhs /; test
```

where the right-hand side is returned only if all conditions evaluate to True. Conditions can be used with **SetDelayed** (**:=**) or **ReplaceDelayed** (**:>**) but generally should not be used with **Set** (**=**) or **Replace** (**→**) because evaluation occurs immediately, after which the condition is no longer present in the definition of the function. Also note that all names appearing in a condition must appear in the pattern to which it is applied; otherwise, use conditional statements **If**, **Which**, or **Switch** as discussed in a later section. Conditions should be applied to the smallest element containing all necessary symbols and can be applied either on the lhs or the rhs. Thus,

```
Clear[f];
f[x_, y_] := x - y /; x > y;
f[x_, y_] := y - x /; x ≤ y
```

successfully computes $|x-y|$ for numerical arguments

```
{f[4, 2], f[0.2, 3], f[x, y]}
```

```
{2, 2.8, f[x, y]}
```

but

```
Clear[g];
g[x_ /; x > y, y_] := x - y ;
g[x_ /; x ≤ y, y_] := y - x
```

```
{g[4, 2], g[0.2, 3], g[x, y]}
```

```
{g[4, 2], g[0.2, 3], g[x, y]}
```

does not because **y** is not present in the pattern **x_**. Nor does it help to apply the condition after **y_** (try it!). On the other hand,

```
Clear[f];
f[x_ /; x > 0] := x;
f[x_ /; x ≤ 0] := -x
```

```
{f[-6], f[0], f[π]}
```

```
{6, 0, π}
```

```
Clear[g];
g[x_ ] := x  /; x > 0;
g[x_ ] := -x  /; x ≤ 0
```

```
{g[-6], g[0], g[π]}
```

```
{6, 0, π}
```

both compute $|x|$ because the condition is formed properly on either side of the assignment.

When possible, it is more efficient to apply the condition to the argument rather than to the rhs because the latter method requires evaluation of each rhs for which a rule exists whereas the former method can be accomplished with fewer evaluations.

Conditions can be combined using logical operators to form a single logical condition that evaluates to either true or false. Composite conditions should be enclosed in parentheses to ensure proper parsing. The following example illustrates some composite conditions. Note that the function remains unevaluated if the truth of condition cannot be determined or if a suitable definition is absent.

```
Clear[f];
f[x_] := "I am a positive even integer" /; (x > 0 && Mod[x, 2] == 0)
f[x_] := "I am a negative odd integer" /; (x < 0 && Mod[x, 2] == 1)
f[x_] := "I am not an integer" /; (Mod[x, 2] ≠ 0 && Mod[x, 2] ≠ 1)
```

```
f /@ {q, -2, 8, -7, π}
```

```
{f[q], f[-2], I am a positive even integer,
 I am a negative odd integer, I am not an integer}
```

Example: step function

The unit step function is defined to be +1 for positive arguments or 0 for negative arguments. By convention, we choose the intermediate value of $\frac{1}{2}$ for the origin. This function can be created using multiple definitions that test the argument.

```
Clear[step];
step[x_ /; x > 0] := 1;
step[x_ /; x == 0] := 1/2;
step[x_ /; x < 0] := 0
```

```
{step[-1/2], step[0], step[12], step[x], step[1 - x] /. x → 0.4}
```

```
{0, 1/2, 1, step[x], 1}
```

All of these cases give the expected output except **step[x]**, which remains unevaluated because the condition is ambiguous, being neither definitely **True** nor definitely **False**.

Note that the condition can be applied to either the left or the right sides of the assignment.

```
Clear[step];
step[x_] := 1  /; x > 0;
step[x_] := 1/2 /; x == 0;
step[x_] := 0  /; x < 0
```

```
{step[-1/2], step[0], step[12], step[x], step[1 - x] /. x → 0.4}
```

```
{0, 1/2, 1, step[x], 1}
```

■ Example: spherical Bessel function at origin

Recall our previous definition for the regular spherical Bessel function.

```
Clear[sbessj];
sbessj[n_, z_] := Sqrt[π/2] z^(-1/2) BesselJ[n + 1/2, z];
```

$$\texttt{sbessj[n\_, z\_]} := \sqrt{\frac{\pi}{2}}\, z^{-1/2}\, \texttt{BesselJ}\left[n + \frac{1}{2}, z\right];$$

Unfortunately, this simple definition cannot handle the origin.

```
{sbessj[0, 0], sbessj[1, 0], sbessj[n, 0]}
```

Power::infy : Infinite expression $\frac{1}{\sqrt{0}}$ encountered. ≫

∞::indet : Indeterminate expression $0 \sqrt{\frac{\pi}{2}}$ ComplexInfinity encountered. ≫

Power::infy : Infinite expression $\frac{1}{\sqrt{0}}$ encountered. ≫

∞::indet : Indeterminate expression $0 \sqrt{\frac{\pi}{2}}$ ComplexInfinity encountered. ≫

Power::infy : Infinite expression $\frac{1}{\sqrt{0}}$ encountered. ≫

General::stop :
  Further output of Power::infy will be suppressed during this calculation. ≫

```
{Indeterminate, Indeterminate, ComplexInfinity}
```

Since we should know the limits for these functions as $x \to 0$, or can deduce them using **Limit**, where we specify that the limit is to be evaluated for decreasing *x*.

```
Limit[sbessj[0, x], x → 0, Direction → -1]
```

```
1
```

```
Limit[sbessj[1, x], x → 0, Direction → -1]
```

```
0
```

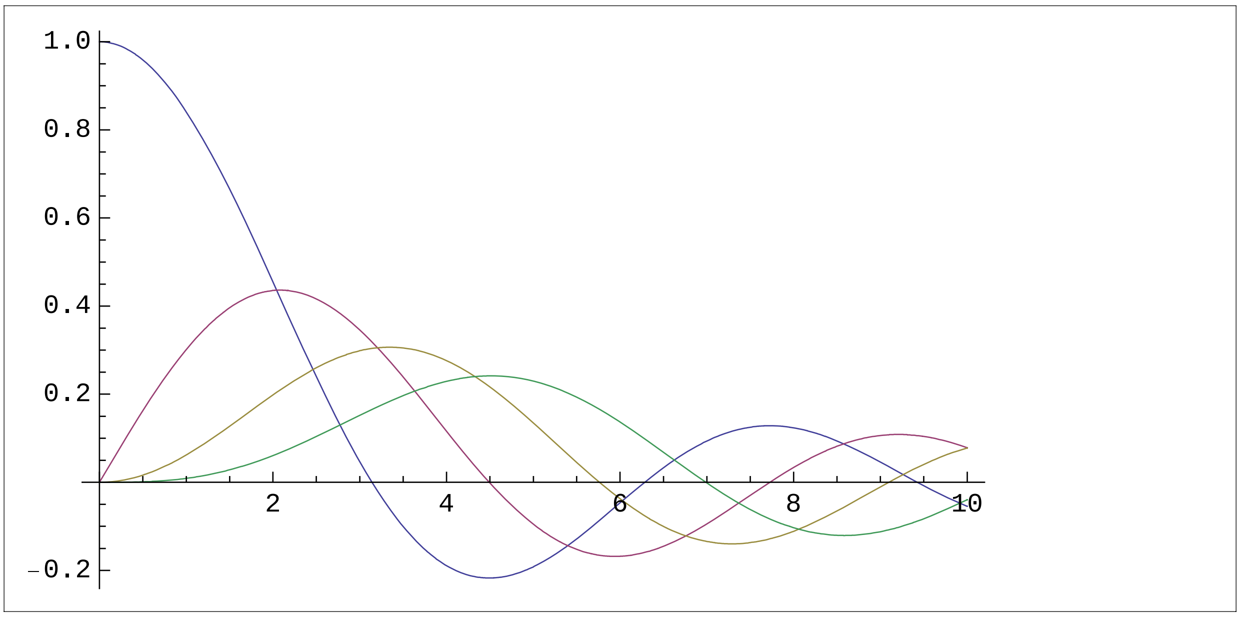
We can include these properties in the definition of the function itself by applying conditions to the arguments and including several definitions covering various special cases.

```
Clear[sbessj];
```

$$\texttt{sbessj[n\_, z\_ /; z} \neq \texttt{0] :=} \sqrt{\frac{\pi}{2}}\, z^{-1/2}\, \texttt{BesselJ}\left[n + \frac{1}{2}, z\right];$$

```
sbessj[0, 0] := 1;
sbessj[n_ /; n > 0, 0 ] := 0
```

```
Plot[Evaluate[Table[sbessj[n, x], {n, 0, 3}]], {x, 0, 10}]
```



This is certainly an improvement, but it is still not as complete as we would like; for example, it does not provide explicit derivatives.

```
D[sbessj[2, z], z]
```

$\texttt{sbessj}^{(0,1)}[2, z]$

■ Exercises

▼ Write and plot a function whose values are $1 - |x|$ for $x \leq 1$ or 0 for $x \geq 1$ without using Abs.

▼ The double factorial function is defined for positive integers as dfact[n] = $n(n-2)\cdots n_{\text{min}}$ where $n_{\text{min}} = 1$ for odd or $n_{\text{min}} = 2$ for even integers. Write a recursive version of this function using conditions that restrict the argument to positive integers. Be sure to test the function. [Hint: you can both specify the argument type and apply a condition to the argument.]

▼ Write a function which returns +1 for positive even integers, -1 for positive odd integers, or 0 for anything else using conditions (do not use If). Be sure to test your function. [Hint: the functions OddQ and EvenQ are helpful.]

▼ Suppose that the earth is modelled by a sphere of uniform density with mass M and radius R. Then, according to Newton's law of gravity, the gravitational force per unit mass, $g(r)$, that acts on a body would be $G\,M \big/ r^2$ for $r \geq R$ or $G\,M\,r \big/ R^3$ for $r \leq R$, where G is the gravitational constant. It is simplest to employ dimensionless units in which {G→1,M→1,R→1}; for any particular case, the scale can be adjusted later by multiplying $g(r)$ by the surface gravity $G\,M \big/ R^2$. Write a function for $g(r)$ and plot it in dimensionless form.

Pattern tests

Some conditions are best applied using *predicates*, which are built-in functions which ask questions that can be answered as either **`True`** or **`False`**. Most of these predicates have function names which end in **`Q`** for question. A list of the available predicates is easily obtained, plus a few interlopers such as **`LegendreQ`** (the Legendre function of the second kind is usually designated Q_n).

```
? *Q
```

▼ System`

AlgebraicIntegerQ	LegendreQ	PositiveDefiniteMatrixQ
AlgebraicUnitQ	LetterQ	PossibleZeroQ
ArgumentCountQ	LinkConnectedQ	PrimePowerQ
ArrayQ	LinkReadyQ	PrimeQ
AtomQ	ListQ	QuadraticIrrationalQ
CoprimeQ	LowerCaseQ	RootOfUnityQ
DigitQ	MachineNumberQ	SameQ
DistributionDomainQ	MatchLocalNameQ	SquareFreeQ
DistributionParameterQ	MatchQ	StringFreeQ
EllipticNomeQ	MatrixQ	StringMatchQ
EvenQ	MemberQ	StringQ
ExactNumberQ	NameQ	SyntaxQ
FreeQ	NumberQ	TensorQ
HermitianMatrixQ	NumericQ	TrueQ
HypergeometricPFQ	OddQ	UnsameQ
InexactNumberQ	OptionQ	UpperCaseQ
IntegerQ	OrderedQ	ValueQ
IntervalMemberQ	PartitionsQ	VectorQ
InverseEllipticNomeQ	PolynomialQ	

▼ PacletManager`

PacletNewerQ

Among these some of the most important are:

```
? IntegerQ
```

IntegerQ[*expr*] gives True if *expr* is an integer, and False otherwise. ≫

```
? EvenQ
```

EvenQ[*expr*] gives True if *expr* is an even integer, and False otherwise. ≫

```
?OddQ
```

OddQ[*expr*] gives True if *expr* is an odd integer, and False otherwise. >>

```
?NumberQ
```

NumberQ[*expr*] gives True if *expr* is a number, and False otherwise. >>

```
?NumericQ
```

NumericQ[*expr*] gives True if *expr* is a numeric quantity, and False otherwise. >>

```
?VectorQ
```

VectorQ[*expr*] gives True if *expr* is a list or a one-dimensional SparseArray object, none of whose elements are themselves lists, and gives False otherwise.
VectorQ[*expr*, *test*] gives True only if *test* yields True when applied to each of the elements in *expr*. >>

```
?MatrixQ
```

MatrixQ[*expr*] gives True if *expr* is a list of lists or a two-dimensional SparseArray object that can represent a matrix, and gives False otherwise.
MatrixQ[*expr*, *test*] gives True only if *test* yields True when applied to each of the matrix elements in *expr*. >>

Unfortunately, there are also several exceptions to the naming convention for predicates.

```
?Positive
```

Positive[*x*] gives True if *x* is a positive number. >>

```
?Negative
```

Negative[*x*] gives True if *x* is a negative number. ≫

Predicates are used to test arguments using the syntax **x_?predicate**. Note that in this context the question mark is interpreted as a **PatternTest**, which should be distinguished from the *information escape* character, with the same appearance, used to obtain interactive help. The information escape cannot be embedded in composite expressions, whereas the pattern test can be used in expressions liberally.

For example, the following function

```
Clear[f];
f[x_?EvenQ] := x^2;
f[x_?OddQ] := -x^3
```

is defined for integers

```
Table[f[n], {n, -5, 5}]
```

```
{125, 16, 27, 4, 1, 0, -1, 4, -27, 16, -125}
```

but is undefined for real numbers.

```
f[1.5]
```

```
f[1.5]
```

A pattern test can be used with a type-specific pattern.

```
Clear[f];
f[x_Integer?Positive] := "I am a positive integer"
f[x_Integer?Negative] := "I am a negative integer"
f[x_Real?Positive] := "I am a positive real number"
f[x_Real?Negative] := "I am a negative real number"
f[x_Symbol] := "I don't know what I am!"
```

```
f /@ {q, 14, -27, π, N[π]}
```

```
{I don't know what I am!,
 I am a positive integer, I am a negative integer,
 I don't know what I am!, I am a positive real number}
```

More generally, **x_?test** is used to test a pattern using a pure function that returns either **True** or **False** when presented with an argument. Several tests can be combined using logical operators. For example, the following function is defined only for arguments which are positive prime numbers.

```
Clear[f, x];
f[x_?(PrimeQ[#] && Positive[#] &)] := 1/x
```

```
f /@ {-4, -3, x, 16, 17, "stuff"}
```

$\left\{f[-4], f[-3], f[x], f[16], \frac{1}{17}, f[stuff]\right\}$

The construction **x_?(⋯)** indicates that the pattern **x_** is tested using the function contained within the parentheses. The function is composed of two logical functions connected by a logical operator, in this case **And** or **&&**, to make a single logical function. Our composite logical function receives a single argument in its slot, #, and is identified as a pure function by the **&** operator. The parentheses are needed with skeletal notation to identify the beginning and end of the pure function. The same effect could have been achieved by defining the test using **Function[var,body]** instead.

▼ Rewrite the function above using conditionals instead of a pattern test.

▼ Write a recursive factorial function which only responds to arguments which are nonnegative integers. Why are these restrictions necessary?

▼ Write a recursive double factorial function for nonnegative integers using a pattern test.

■ Example: a function which tests orthogonality

For example, suppose that we need a function that tests for the orthogonality of a matrix. A matrix is orthogonal if when multiplied by its transpose an identity matrix is obtained. Since orthogonality is meaningless for other objects, we use the predicate **MatrixQ** to restrict the definition to matrices. We verify that the matrix is square using **Dimensions** and compare the product with its transpose to the identity matrix of the same length.

```
Clear[testOrthogonality];

testOrthogonality[matrix_?MatrixQ] :=
  MatchQ[Dimensions[matrix], {x_, x_}] &&
   matrix . Transpose[matrix] == IdentityMatrix[Length[matrix]];

testOrthogonality[x_?NumericQ] := x^2 == 1
```

In the following cases our function successfully identifies a manifestly orthogonal matrix, rejects a matrix that is not square, and does not venture a guess about a symbol of uncertain character.

testOrthogonality$\left[\left\{\left\{\frac{1}{\sqrt{2}}, \frac{1}{\sqrt{2}}\right\}, \left\{-\frac{1}{\sqrt{2}}, \frac{1}{\sqrt{2}}\right\}\right\}\right]$

```
True
```

```
testOrthogonality[{{1/Sqrt[2], 1/Sqrt[2], 0}, {-1/Sqrt[2], 1/Sqrt[2], 1}}]
```

$$\text{testOrthogonality}\left[\left\{\left\{\frac{1}{\sqrt{2}}, \frac{1}{\sqrt{2}}, 0\right\}, \left\{-\frac{1}{\sqrt{2}}, \frac{1}{\sqrt{2}}, 1\right\}\right\}\right]$$

```
False
```

```
testOrthogonality[x]
```

```
testOrthogonality[x]
```

We also included a definition to handle expressions which evaluate to numbers, using **NumericQ** rather than **NumberQ**. (Convince yourself that **NumberQ** is not up to the task!)

$$\text{testOrthogonality /@}\left\{1,\ 2,\ -1,\ I,\ \frac{1}{\sqrt{2}}(1+I),\ \text{Cos}[0],\ \text{Cos}[0.25]\right\}$$

```
{True, False, True, False, False, True, False}
```

A slightly more difficult case:

$$\text{testOrthogonality@}\begin{pmatrix} \frac{1}{3} & \frac{2\sqrt{2}}{3} & 0 \\ \frac{2}{3} & -\frac{1}{3\sqrt{2}} & \frac{1}{\sqrt{2}} \\ -\frac{2}{3} & \frac{1}{3\sqrt{2}} & \frac{1}{\sqrt{2}} \end{pmatrix}$$

```
True
```