10 Front end programming

In this chapter we extend the programming concepts we have covered thus far to the objects that comprise the user interface, or front end. Because the objects that the *Mathematica* user interacts with are themselves *Mathematica* expressions, all of the tools that you use to do computations can also be used to create, manipulate, and alter cells and notebooks themselves. We will first look at the underlying structure of these objects and then discuss ways of manipulating them directly from within *Mathematica*.

10.1 Introduction

Up until this point, we have been primarily concerned with learning about programming constructs and styles so that we can write programs to manipulate data or solve problems from science, engineering, or mathematics. We have taken for granted that the space in which we do our experimenting, prototyping, and documenting has been the *Mathematica* notebook, an interface that has some similarities to a word processor document.

It is not uncommon now to add interactive elements to your documents to make them more useful for yourself or the intended reader of your documents. With programs, documentation, and papers all being created and used in electronic format, *Mathematica* provides a seamless and well-integrated interface to these elements.

Another tool that is useful, especially for educators, are buttons that allow you to hide your program code behind a familiar and easy-to-use interface element – the button. The user clicks on a button and an action happens that is determined by the underlying code. For example, you might want to have calculus students quickly plot Taylor polynomial approximations to a function together with the original function but do not want them to spend time learning the syntax of such commands in *Mathematica*. You could easily program an interface that would only require them to fill in a few parameters before clicking a button to produce the desired plot.

In this chapter we will discuss the structure of cell and notebook expressions, look at a few basic functions for manipulating these expressions, and then create several simple examples that give a flavor of the kind of things that can be done with front end programming.

Before we begin we should mention that this chapter is not intended as a complete discussion of front end programming. An entire book could certainly be written on this topic alone. This book is intended to give you an introduction to the many aspects of programming with *Mathematica* and front end programming is certainly an appropriate topic for that introduction. But there are several areas that cannot be included here, either because of space limitations or because they do not fit under the introductory nature of this book. These topics include front end options and front end tokens. An understanding of each of these topics is quite important for more advanced front end programming. The interested reader can delve further into this subject by looking in the Front End category of the Help Browser or by searching the *Mathematica* Information Center online at library.wolfram.com/infocenter.

10.2 The structure of cells and notebooks

We have spent a lot of time in this book focusing on the structure of *Mathematica* expressions. In Chapter 2 we indicated that *Mathematica* expressions are of the form `h[`e_1`, `e_2`, ...]` where `h` is the head of the expression and the e_i are the elements which may themselves be *Mathematica* expressions. We even went so far as to say that everything in *Mathematica* is an expression. In this section we will learn that this statement extends to elements of the front end, specifically to notebooks and cells.

Notebook expressions

Notebooks are ASCII files, meaning that you can open them in a text editor and view their contents directly. If you were to do that, you would see that the underlying expression is a *Mathematica* function called a `Notebook`. The notebook would look like this:

```
Notebook[{
    Cell[string, style, options],
    Cell[string, style, options],
    …
    },
  options]
```

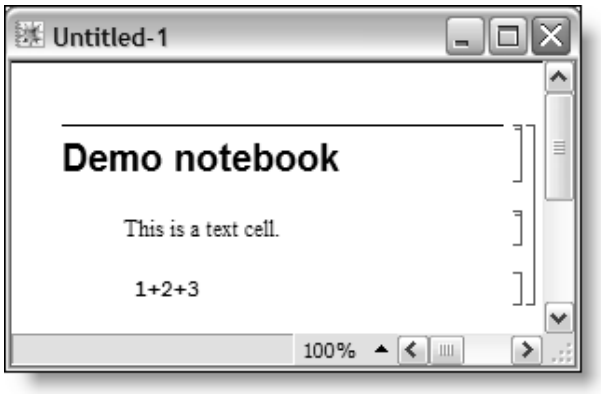
In other words, the `Notebook` is a function whose first argument is a list of one or more `Cell` objects, followed by some options. The *Mathematica* kernel does not do

anything with this practically. It is the *Mathematica* front end that knows how to render this expression as the familiar notebook.

For example, here is a very simple notebook that you could write in a text editor (of course there is no reason to do that).

```
Notebook[{
    Cell["Demo notebook", "Section"],
    Cell["This is a text cell.","Text"],
    Cell["1+2+3", "Input"]
    }]
```

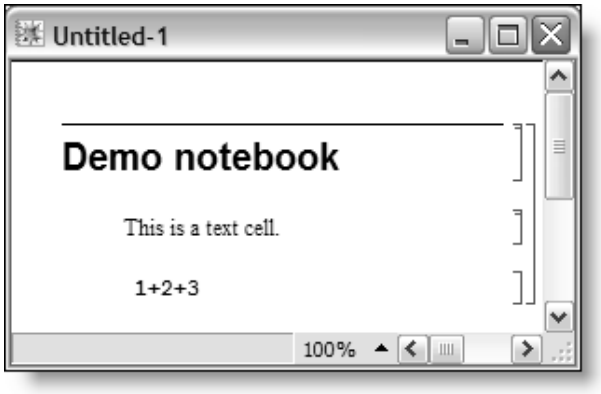
The *Mathematica* front end renders this expression in the familiar manner, a window.



Let us create the notebook from scratch using a kernel command, `NotebookPut`. `NotebookPut[`*expr*`]` will create a notebook corresponding to *expr* in the front end and make it the currently selected notebook.

```
In[1]:= nb = NotebookPut[
           Notebook[{
             Cell["Demo notebook", "Section"],
             Cell["This is a text cell", "Text"],
             Cell["1+2+3", "Input"]
            }]
          ]
Out[1]= NotebookObject[ <<Untitled-1>> ]
```

Here is the notebook as viewed in the front end.



There is actually quite a lot going on behind the scenes here in terms of the interaction between the kernel and the front end. As stated in Chapter 1, the kernel and the front end are two separate programs that communicate with each other through a protocol called *MathLink*. For purposes of efficiency, *MathLink* itself does not store the notebook in memory but instead refers to it by means of a *handle*. These handles are called *notebook objects* and are given as `NotebookObject[`*fe*`,`*id*`]`, where *fe* is an object that refers to the entire front end and *id* is an integer that is a unique identifier for that notebook. In the example above, looking at the `InputForm` displays this information stored with the notebook object.

```
In[2]:= InputForm[nb]

Out[2]//InputForm=
NotebookObject[FrontEndObject[LinkObject["3v8_shm",
   1, 1]], 28]
```

Since we have assigned a symbol, `nb`, to this object, we can refer to it through this symbol. `NotebookGet` gets the expression corresponding to this notebook and reads it into the kernel. You should think of it as analogous to `Get` for packages.

```
In[3]:= NotebookGet[nb]

Out[3]= Notebook[
         {Cell[CellGroupData[{Cell[Demo notebook, Section], Cell[
              This is a text cell, Text], Cell[1+2+3, Input]}, Open]]},
         FrontEndVersion → 5.0 for Microsoft Windows,
         ScreenRectangle → {{0., 1024.}, {0., 681.}}]
```

Notice that the front end has added two options to this notebook: `FrontEndVersion` and `ScreenRectangle`. It has also added some grouping information for the cells.

These are default behaviors of the front end and may vary from one front end to another. They are also user-settable.

Manipulating notebooks

`NotebookPut` and `NotebookGet` are general functions for dealing with entire notebooks at once. There are a host of additional functions for manipulating parts of notebooks. You might first think that we can simply use functions like `Part` to extract a particular part of a notebook we are interested in. There are several reasons why this is not generally practical. First, because a notebook can contain many, many cells, it is often quite difficult to determine precisely which part you want to work on. Secondly, since the notebook resides in the front end, not the kernel, it is often not very efficient to manipulate the notebook directly by the kernel (although, if the notebook is small enough, this is certainly possible).

As it turns out, there is a way around these issues and that is through something referred to as the "current selection," which is essentially a reference to the notebook object. You could then think of the notebook manipulation functions as operating on streams.

To see a list of the open notebooks, use `Notebooks[]`.

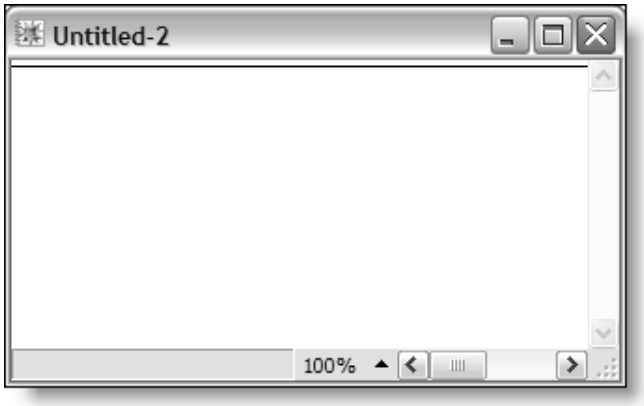
```
In[4]:= Notebooks[]

Out[4]= {NotebookObject[ <<Untitled-1>> ],
         NotebookObject[ <<10FEProgramming.nb>> ],
         NotebookObject[ <<Messages>> ]}
```

Again, using `InputForm`, you can see the actual handles to each of the notebooks.

```
In[5]:= Notebooks[] // InputForm

Out[5]//InputForm=
{NotebookObject[FrontEndObject[LinkObject["3v8_shm",
    1, 1]], 28], NotebookObject[
  FrontEndObject[LinkObject["3v8_shm", 1, 1]], 27],
 NotebookObject[FrontEndObject[LinkObject["3v8_shm",
    1, 1]], 7]}
```

Let us walk through some of the most common notebook operations you should learn about. The first is `NotebookCreate`. As its name implies, this function will create a new untitled notebook in the front end. We assign `nb` to be the handle to this notebook.

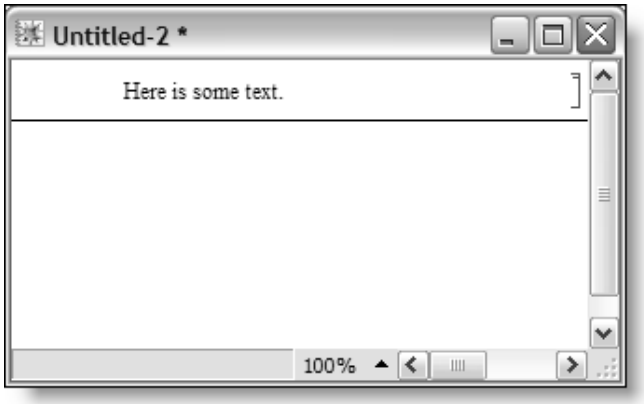
```
In[6]:= nb = NotebookCreate[]
Out[6]= NotebookObject[ <<Untitled-2>> ]
```



Now let us write to the notebook. `NotebookWrite` takes two arguments: the first argument is the notebook object that we are writing to; the second argument is what we are writing. We will create a few different examples below.

A `Cell` is an expression with two arguments. The first argument is the contents of the cell; the second argument is the cell style, a listing of which is under the Format▹Style menu in the front end.

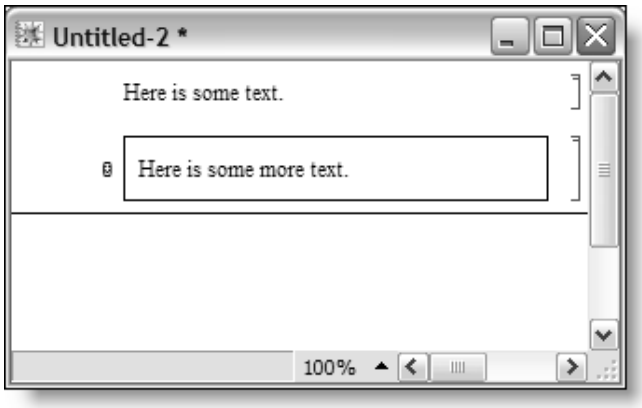
```
In[7]:= NotebookWrite[nb, Cell["Here is some text.", "Text"]]
```



Adding options to `Cell` allows us to change some of the properties of the cell. For example, here are several of the options that you can add.

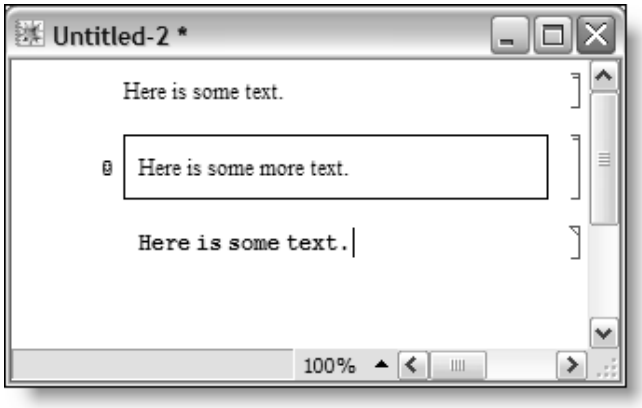
```
In[8]:= Take[Options[Cell], {10, 15}]

Out[8]= {Deletable → True, PageWidth → WindowWidth, Visible → True,
  CellFrame → False, CellDingbat → None, ShowCellBracket → True}
```

```
In[9]:= NotebookWrite[nb,
  Cell["Here is some more text.", "Text",
   CellFrame → True, CellDingbat → ☺]]
```



If we simply give a string as the second argument to `NotebookWrite`, *Mathematica* will use the default cell type, `Input`.

```
In[10]:= NotebookWrite[nb, "Here is some text."]
```



Now suppose we wanted to insert an input cell with the expression 2^{100} in it.

```
In[11]:= 2^100

Out[11]= 1267650600228229401496703205376
```

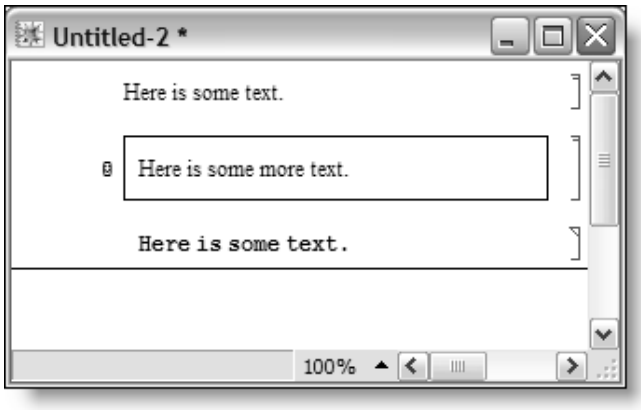
If you were to look at the underlying expression of the above cell (under the Format menu, choose Show Expression), it would look like this:

```
Cell[BoxData[
    SuperscriptBox["2", "100"]], "Input"]
```

We will talk about `BoxData` in just a moment, but we should be able to insert a cell like this directly into our notebook object. Before we do this, notice that the insertion point has been left inside the Input cell after the last `NotebookWrite`. To move the cell insertion bar after the current cell, we will use `SelectionMove` which takes three arguments: the notebook we are operating on, the direction to move, and the unit by which we should move. The direction can be any of `Next`, `Previous`, `After`, `Before`, `All`. The units are things like `Word`, `Cell`, `CellGroup`, `Notebook` (see the Help Browser under `SelectionMove` for a complete description).

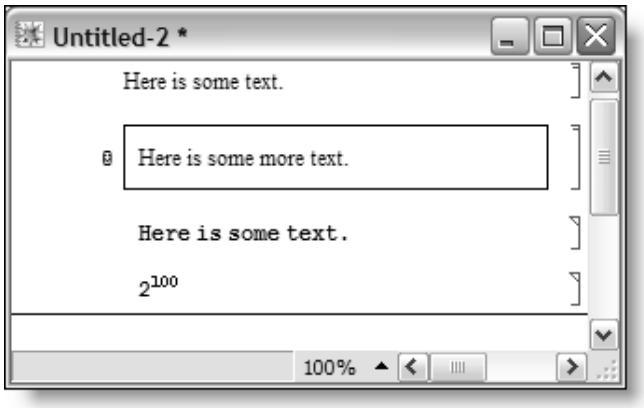
So, in our example, we want to move the selection just after the present cell.

```
In[12]:= SelectionMove[nb, After, Cell]
```



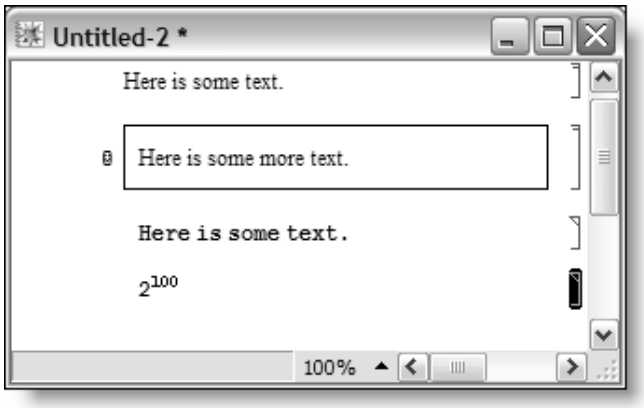
Now we can write the input cell to the notebook.

```
In[13]:= NotebookWrite[nb,
          Cell[BoxData[SuperscriptBox["2", "100"]], "Input"]]
```



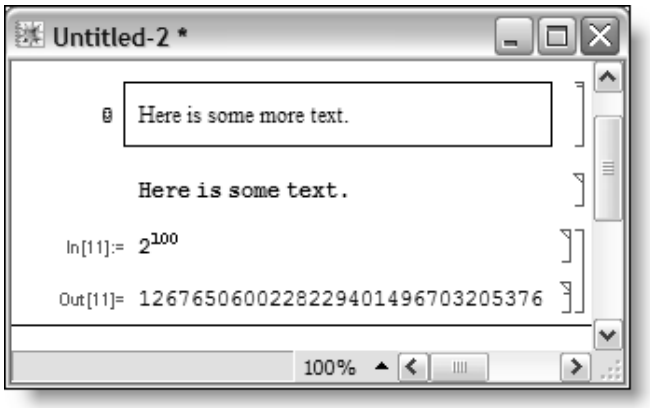
Notice that at the end of each `NotebookWrite`, the cell insertion bar was placed just after the cell that was written, except in the case of writing input cells. Oftentimes, you will need to move around within the notebook or select a particular cell (or other expression) and perform some operation on it. For example, suppose we would like to select the previous cell (the one containing the 2^{100}) in `nb` and evaluate it. We can do this with the `SelectionMove` function.

```
In[14]:= SelectionMove[nb, Previous, Cell]
```



To evaluate the currently selected expression, use `SelectionEvaluate`.

```
In[15]:= SelectionEvaluate[nb]
```



Let us put a few of these pieces together and create a function that will evaluate the next input cell. In Section 10.5 we will turn this code into a button.

For this example we will operate in the current notebook. We can refer to the notebook in which these commands are being evaluated by `EvaluationNotebook[]`. First we select the current unit; that is, the cell in which the following code lives.

```
SelectionMove[EvaluationNotebook[],All,Cell]
```

Then we move the selection insertion to the next cell (at the moment, this code will only work if it is immediately followed by an input cell).

```
SelectionMove[EvaluationNotebook[],Next,Cell]
```

Finally, we evaluate the currently selected input.

```
SelectionEvaluate[EvaluationNotebook[]]
```

Here we bundle this code up into the function `EvaluateNext`.

```
In[16]:= EvaluateNext := (
           SelectionMove[EvaluationNotebook[], All, Cell];
           SelectionMove[EvaluationNotebook[], Next, Cell];
           SelectionEvaluate[EvaluationNotebook[]];
         )
```

Evaluating the cell containing `EvaluateNext` causes the immediately following cell to be evaluated.

```
In[17]:= EvaluateNext

In[18]:= 2 + 2

Out[18]= 4
```

Exercises

1. Using `NotebookPut`, create a notebook with one Title cell, one Section cell, one Text cell and two Input cells.
2. Use `NotebookGet` to read the notebook you created in Exercise 1 into the kernel. Then programmatically change the Section cells to Subsection cells either using `Cases` or an appropriate rule.
3. Take either of the notebooks you created in the above exercises and use `SelectionMove` and `SelectionEvaluate` to evaluate all of the Input cells in the notebook.

10.3 Cell data types

The cells in your notebooks often contain different kinds of data. Sometimes they will only contain text. Other times they may contain formatted mathematical expressions, or possibly a graphical object. Since the `Cell` data object has to handle each one of these kinds of data, there is a mechanism that enables the front end to deal with these objects in a consistent manner – cell data types. We will look at a few of the most important and useful cell data types in the next few sections.

TextData

Let us first look at a text cell that contains no special formatting.

```
Cell["Here is some text.", "Text"]
```

The formatted version of this cell looks like this:

Here is some text.

Adding some formatting to this cell causes a `TextData` wrapper to be added.

```
Cell[TextData[{
  "Here is some ", StyleBox["italicized", FontSlant->"Italic"],
  " text."
}], "Text"]
```

The formatted version of this cell looks like this:

Here is some *italicized* text.

Cells with `TextData` can contain a number of other data objects embedded in the cell. For example, here is a text cell that contains a `ValueBox`. `ValueBox`es provide a means of embedding evaluations inside of your text cells.

```
Cell[TextData[{
  "The current version is: ", ValueBox["$Version"]
}], "Text"]
```

The formatted version of this cell looks like this:

The current version is: 5.1 for Microsoft Windows

A listing of all of the possible `ValueBox` names that can be used can be found choosing Create Value Display Object from the Input menu. Looking under the list of global variables that can be used as the argument to `ValueBox`, you will see `Date`, for example.

```
Cell[TextData[{
  "The current date is: ",
  ValueBox["DateLong"]
}], "Text"]
```

The formatted version of this cell looks like this:

The current date is: Sunday, September 5, 2004

BoxData

Many of your cells in *Mathematica* will contain formatted mathematical expressions. Whenever you work with these two-dimensional typeset objects, a different editor is invoked, called the math editor. This is indicated in the front end by a pink background in `Text` cell style on the typeset expression (you can enter a math typeset expression by pressing Control-9). This is also indicated in the underlying cell structure by means of the `BoxData` wrapper. For example, consider the following cell containing a superscript expression.

```
Cell[BoxData[
    RowBox[{
      SuperscriptBox["x", "2"], "+", "y"}]], "Input"]
```

The formatted version of this cell looks like this:

$x^2 + y$

There are several things to note here. First, we see that *Mathematica* has automatically placed the elements x^2, + and y all in something called a `RowBox`. This is how *Mathematica* represents box objects or a series of strings.

Secondly, the x^2 object is represented internally as another box object, specifically `SuperscriptBox[x,2]`. You can use `DisplayForm` to print box expressions in an explicit two-dimensional form.

```
In[1]:= SuperscriptBox[x, 2] // DisplayForm

Out[1]//DisplayForm=
      x^2
```

There are many different box objects in *Mathematica*. Below are just a few commonly used box objects.

```
Cell[BoxData[
    SqrtBox["2"]], "Input"]
```

The formatted version of this cell looks like this:

$$\sqrt{2}$$

```
Cell[BoxData[
    FractionBox["x", "y"]], "Input"]
```

The formatted version of this cell looks like this:

$$\frac{x}{y}$$

```
Cell[BoxData[
    RowBox[{
      SubsuperscriptBox["∫", "a", "b"],
      RowBox[{"x", " ",
        RowBox[{"d", "x"}]
      }]
    }]
  ], "Input"]
```

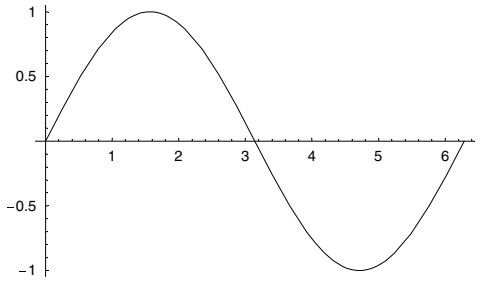
The formatted version of this cell looks like this:

$$\int_a^b x\,dx$$

GraphicsData

Another type of data wrapper that you will encounter is `GraphicsData`, used to indicate a graphical object in the cell. For example, creating a graphics object in the front end displays a plot.

```
In[2]:= Plot[Sin[x], {x, 0, 2 π}];
```



If you unformat the graphics cell, the first few lines would look like the following:

```
Cell[GraphicsData["PostScript", "\<\
%!
%%Creator: Mathematica
%%AspectRatio: .61803
MathPictureStart
...
```

Normally you will not create graphics objects from scratch so it would seem as if there is not too much you could do with `GraphicsData` objects manually. But suppose you were interested in displaying your graphics to a notebook other than the one in which you evaluate the graphics input. For example, we could use `NotebookPut` to write out a new notebook containing a graphics cell object as follows:

```
In[3]:= MyDisplayChannel[gr_] :=
          NotebookPut[Notebook[{Cell[GraphicsData[
            "PostScript", DisplayString[gr]], "Graphics"]}]]
```

This is now used by giving `MyDisplayChannel` as the value of `DisplayFunction` for any plot your create.

```
In[4]:= Plot3D[Sin[x y], {x, 0, 2 π}, {y, 0, 2 π},
         DisplayFunction → MyDisplayChannel]

Out[4]= NotebookObject[ <<Untitled-5>> ]
```

Evaluating the above expression will cause a new notebook window to be created in your front end containing just the output of the `Plot3D` command, a graphic of the surface $\sin(x\,y)$.

Exercises

1. Using `NotebookPut`, create a notebook with several Text cells each containing a `ValueBox` such as `$Version`, `$OperatingSystem`, and `$UserName`.
2. Using `NotebookPut`, create a notebook with an Input cell containing the integral $\int \frac{1}{1-x^3}\, dx$. Then evaluate the integral using `SelectionMove` and `SelectionEvaluate`.

10.4 GridBoxes

ShowTable

Whenever you create a two-dimensional expression consisting of some number of rows and columns, *Mathematica* represents that expression as a `GridBox` object. For example, if you used the BasicInput palette to create a 2×2 matrix, it would be represented as follows:

```
Cell[BoxData[GridBox[{
        {"a", "b"},
        {"c", "d"}
        }]], "Input"]
```

The formatted version of this cell looks like this:

$$\begin{matrix} a & b \\ c & d \end{matrix}$$

Looking at the `GridBox` object, you should see that it is identical (structurally) to a matrix in *Mathematica*, which is really just a list of lists.

```
In[1]:= FullForm[{{a, b}, {c, d}}]

Out[1]//FullForm=
List[List[a, b], List[c, d]]

In[2]:= {{a, b}, {c, d}} // MatrixForm

Out[2]//MatrixForm=
a  b
c  d
```

Using GridBoxes, let us create a function for displaying arrays of data in a formatted table. First we create some sample data.

```
In[3]:= data = {{"α", "β", "γ"},
          {1.234, 2.3451, 3.4567801}, {SqrtBox["π"], "x/y", "Γ(n)"}};
```

We can put this data into a GridBox and immediately print it in a two-dimensional grid using DisplayForm.

```
In[4]:= GridBox[data] // DisplayForm
```

```
Out[4]//DisplayForm=
      α       β          γ
  1.234  2.3451  3.4567801
   √π      x/y       Γ(n)
```

GridBox can be given several options that control its appearance.

```
In[5]:= Options[GridBox]
```

```
Out[5]= {GridBaseline → Axis, RowSpacings → 1., ColumnSpacings → 0.8,
         ColumnWidths → Automatic, RowAlignments → Baseline,
         ColumnAlignments → {Center}, GridFrame → False,
         GridFrameMargins → {{0.4, 0.4}, {0.5, 0.5}},
         RowLines → False, ColumnLines → False, RowMinHeight → 1.,
         RowsEqual → False, ColumnsEqual → False,
         AutoDelete → True, AllowScriptLevelChange → True,
         MultilineFunction → None, GridDefaultElement :→ □}
```

Let us add a frame, make the margins around each grid element a bit larger than the default, and add some lines between the rows and columns. Usually you will set the values for GridFrame, RowLines, and ColumnLines to either True or False to enable or disable these elements. Giving an explicit number as the value of each of these options gives the thickness of the line that is drawn for that object.

```
In[6]:= GridBox[data,
          GridFrame → 1.2, GridFrameMargins → {{1, 1}, {1, 1}},
          RowLines → 1, ColumnLines → 1] // DisplayForm
```

```
Out[6]//DisplayForm=
```

α	β	γ
1.234	2.3451	3.4567801
√π	x/y	Γ(n)

Now we can bundle up this code and turn all of it into a function, ShowTable. If we wish, we can add some formatting, but to do so we have to wrap the GridBox in a Style:

`Box`. `FontSize`, `FontFamily`, `Background`, and `SingleLetterItalics` are all options to `StyleBox`.

```
In[7]:= ShowTable[data_] := DisplayForm[StyleBox[
          GridBox[data,
           GridFrame → 1.2, GridFrameMargins → {{1, 1}, {1, 1}},
           RowLines → 1, ColumnLines → 1],
          FontFamily → "Times",
          Background → GrayLevel[.8], SingleLetterItalics → True
         ]]

In[8]:= ShowTable[data]
```

Out[8]//DisplayForm=

α	β	γ
1.234	2.3451	3.4567801
$\sqrt{\pi}$	$\frac{x}{y}$	$\Gamma(n)$

Sometimes the data you work with will need to be manipulated in some way to display it. The following is another example of the use of `ShowTable`, but one for which we first need to think about the dimensions of our data. Consider displaying a table of reciprocals of rep units, numbers consisting entirely of 1s.

```
In[9]:= RepUnit[n_?Positive] := Nest[10 #1 + 1 &, 1, n - 1]

In[10]:= expr = Map[1/RepUnit[#] &, Range[12]]
```

Out[10]=

$$\left\{1, \frac{1}{11}, \frac{1}{111}, \frac{1}{1111}, \frac{1}{11111}, \frac{1}{111111}, \frac{1}{1111111}, \frac{1}{11111111}, \frac{1}{111111111}, \frac{1}{1111111111}, \frac{1}{11111111111}, \frac{1}{111111111111}\right\}$$

Since the above output contains 12 expressions, we need to explicitly partition it to be rectangular. First we partition the data into rows of three elements (columns) each.

```
In[11]:= ShowTable[Partition[expr, 3]]
```

Out[11]//DisplayForm=

1	$\frac{1}{11}$	$\frac{1}{111}$
$\frac{1}{1111}$	$\frac{1}{11111}$	$\frac{1}{111111}$
$\frac{1}{1111111}$	$\frac{1}{11111111}$	$\frac{1}{111111111}$
$\frac{1}{1111111111}$	$\frac{1}{11111111111}$	$\frac{1}{111111111111}$

Here we partition the data into rows of four elements each.

```
In[12]:= ShowTable[Partition[expr, 4]]
```

Out[12]//DisplayForm=

1	$\frac{1}{11}$	$\frac{1}{111}$	$\frac{1}{1111}$
$\frac{1}{11111}$	$\frac{1}{111111}$	$\frac{1}{1111111}$	$\frac{1}{11111111}$
$\frac{1}{111111111}$	$\frac{1}{1111111111}$	$\frac{1}{11111111111}$	$\frac{1}{1111111111111}$

In the above tables, we are manually partitioning the rows and columns into sublists that will be rectangular when they are put into the table. It would be good programming style to take that task from the user and do it automatically. We leave this as an exercise.

TriangleForm

In this section we will use `GridBox` to develop a function for displaying an array in a triangular format. Such a function is quite useful for displaying the elements of Pascal's triangle in the familiar triangular array.

```
                                1
                             1      1
                          1      2      1
                       1      3      3      1
                    1      4      6      4      1
                 1      5      10     10     5      1
              1     6      15     20     15     6     1
           1     7     21     35     35     21     7     1
        1     8     28     56     70     56     28     8     1
```

First let us create a function for generating the first *n* rows of Pascal's triangle.

```
In[13]:= PascalTable[rows_] :=
          Table[Binomial[n, m], {n, 0, rows}, {m, 0, n}]
```

Here are the first four rows (including the 0th row).

```
In[14]:= expr = PascalTable[3]
Out[14]= {{1}, {1, 1}, {1, 2, 1}, {1, 3, 3, 1}}
```

If we put empty strings around the elements in the appropriate places we can see what the grid should look like.

```
In[15]:= GridBox[{
           {"", "", 1, "", ""},
           {"", 1, "", 1, ""},
           {1, "", 2, "", 1}
         }] // DisplayForm
```

```
Out[15]//DisplayForm=
          1
        1   1
      1   2   1
```

So we need to develop a function to insert these empty strings between each element in each row and we also need to pad out each row to the length of the longest row in the entire table. First we write the function to pad each row.

```
In[16]:= pad[lis_] := PadLeft[lis, 2 Length[expr] - 1,
           "", Round[(2 Length[expr] - 1 - Length[lis]) / 2]]

In[17]:= pad[expr[[1]]]

Out[17]= {, , , 1, , , }

In[18]:= pad[expr[[2]]]

Out[18]= {, , , 1, 1, , }
```

Now to insert the appropriate number of empty strings between elements, let us first manually insert space in a few rows.

```
In[19]:= Insert[expr[[2]], "", {{2}}]

Out[19]= {1, , 1}

In[20]:= Insert[expr[[3]], "", {{2}, {3}}]

Out[20]= {1, , 2, , 1}

In[21]:= Insert[expr[[4]], "", {{2}, {3}, {4}}]

Out[21]= {1, , 3, , 3, , 1}
```

Here is the function to create the third argument for `Insert`.

```
In[22]:= Map[List, Rest[Range[Length[{1, 3, 3, 1}]]]]

Out[22]= {{2}, {3}, {4}}
```

Here is the function to add the appropriate amount of space between elements in each row.

```
In[23]:= addspace[lis_] :=
          Insert[lis, "", Map[List, Rest[Range[Length[lis]]]]]

In[24]:= addspace[expr[[3]]]

Out[24]= {1, , 2, , 1}

In[25]:= addspace[expr[[1]]]

Out[25]= {1}
```

This maps the `addspace` function across each row of the Pascal table.

```
In[26]:= expr = Map[addspace, PascalTable[3]]

Out[26]= {{1}, {1, , 1}, {1, , 2, , 1}, {1, , 3, , 3, , 1}}
```

Then we pad out each row using our `pad` function developed above.

```
In[27]:= Map[pad, expr]

Out[27]= {{, , , 1, , , }, {, , 1, , 1, , },
          {, 1, , 2, , 1, }, {1, , 3, , 3, , 1}}
```

Finally we put this expression into a `GridBox` and display it.

```
In[28]:= GridBox[%] // DisplayForm
```

```
Out[28]//DisplayForm=
            1
          1   1
        1   2   1
      1   3   3   1
```

Here is the `TriangleForm` function then consisting of the above pieces.

```
In[29]:= TriangleForm[lis_List] :=
          Module[{addspace, expr, len = Length[lis]},
           addspace[l_] :=
            Insert[l, "", Map[List, Rest[Range[Length[l]]]]];
           expr = Map[addspace, lis];
           DisplayForm[GridBox[Map[PadLeft[#, 2 len - 1,
                 "", Round[1/2 (2 len - 1 - Length[#])]] &, expr]]
           ]
          ]
```

```
In[30]:= PascalTable[5] // TriangleForm
```

Out[30]//DisplayForm=

```
                1
              1   1
            1   2   1
          1   3   3   1
        1   4   6   4   1
      1   5   10  10  5   1
```

Exercises

1. Modify `ShowTable` so that it can display a user-specified heading in the first row of the grid. Include formatting to set the style of the strings in the heading to be different than the rest of the elements displayed by `ShowTable`.

2. Modify `ShowTable` so that it automatically partitions the list it is passed to be rectangular, with the number of rows and columns as close to each other as possible.

3. Create a function `TruthTable[`*expr*`,` *vars*`]` that displays the logical expression *expr* together with all the possible truth values for the variables in the list *vars*. For example, here is the truth table for the expression $(A \vee B) \Rightarrow C$.

```
In[1]:= TruthTable[Implies[A || B, C], {A, B, C}]
```

Out[1]//DisplayForm=

A	B	C	$(A \vee B) \Rightarrow C$
T	T	T	T
T	T	F	F
T	F	T	T
T	F	F	F
F	T	T	T
F	T	F	F
F	F	T	T
F	F	F	T

You will first need to create a list of all possible truth value assignments for the variables, `A`, `B`, `C` in this case. One approach would be to use `Distribute`. So, essentially, this is the left-hand side, or first three columns of the above table (not counting the first row containing the table headings).

```
In[2]:= vars = {A, B, C};
        len = Length[vars]; ins =
         Distribute[Table[{True, False}, {len}], List, List, List]

Out[3]= {{True, True, True}, {True, True, False}, {True, False, True},
         {True, False, False}, {False, True, True}, {False, True, False},
         {False, False, True}, {False, False, False}}
```

You can then create a list of rules associating each of these triples of truth values with a triple of variables.

```
In[4]:= Map[Thread[vars → #1] &, ins]

Out[4]= {{A → True, B → True, C → True},
         {A → True, B → True, C → False}, {A → True, B → False, C → True},
         {A → True, B → False, C → False}, {A → False, B → True, C → True},
         {A → False, B → True, C → False}, {A → False, B → False, C → True},
         {A → False, B → False, C → False}}
```

Substituting these rules into the logical expression produces a truth value for each of the above rows.

```
In[5]:= Implies[A || B, C] /. Map[Thread[vars → #1] &, ins]

Out[5]= {True, False, True, False, True, False, True, True}
```

Your task is to put all these pieces together in a `GridBox` with appropriate formatting.

10.5 Buttons

Buttons are very user-friendly objects whose functionality is familiar to any computer user. From the programmer's point of view, they allow you to hide your code behind a graphical element, the button. Instead of writing a function and evaluating it by pressing Shift-Enter from the keyboard, you pass the mouse cursor over the button and simply click. Whatever code is hidden underneath the button is then evaluated.

In this section we will first look at the structure of `ButtonBox`es and then create some examples to demonstrate the variety of tasks that can be accomplished with buttons.

Making buttons the easy way

The simplest way to create buttons is to select an expression in your *Mathematica* notebook, choose Create Button from the Input menu, and then activate your button. Let us walk through these steps to create a button that pastes an expression into your notebook.

Suppose you were writing a paper in which you are discussing sequences and you need to use an expression such as the following repeatedly in your notebook: $\{a_1, a_2, \ldots, a_n\}$. To create a button that would allow you to paste this expression into your notebook by simply clicking that button, we first write down the expression we will work with below in a regular input cell.

```
{a1, a2, …, an}
```

Now select the entire expression and choose Create Button ▹ Paste from the Input menu.

```
{a1, a2, …, an}
```

Finally, to activate the button so that you can click it to have an action occur, select the cell in which the button occurs and then choose Cell Properties ▹ Cell Active from the Cell menu.

```
{a1, a2, …, an}
```

Clicking the above button will paste the following at the insertion point: $\{a_1, a_2, \ldots, a_n\}$.

If you wished, you could create a free-standing palette from this button by choosing Generate Palette from Selection from the File menu.

Although the above procedure for creating buttons is quite straightforward, it is only convenient for fairly simple buttons. For more complicated buttons you will find that you need a good understanding of the structure of buttons and the various options that control their actions and display. We turn to those topics in the next few sections.

The structure of buttons

Buttons are created with the `ButtonBox` function in *Mathematica*. `ButtonBox` takes one argument and by default, that argument is pasted at the current selection point.

In the examples that follow, we will use `DisplayForm` to display the button as an interactive element. If you were to unformat your button (Show Expression from the Format menu), you would see essentially all that precedes the `DisplayForm` below.

```
In[1]:= ButtonBox["some text", Active → True] // DisplayForm
```

```
Out[1]//DisplayForm=
```

some text

Note that we have added the option `Active→True`. This makes the resulting button uneditable, one that is clickable. You will need to add this option to all your buttons to activate them. Clicking this button causes the following to be pasted at the current selection point.

```
some text
```

Let us create a button that can serve as a template for a definite integral.

```
In[2]:= ButtonBox["Integrate[fun,{x,xmin,xmax}]", Active->True]
        //DisplayForm
```

```
Out[2]//DisplayForm=
```

Integrate[fun, {x, xmin, xmax}]

Clicking the button causes the following to be pasted in.

```
Integrate[fun, {x, xmin, xmax}]
```

We can use placeholders in our template button so that the user can move from one placeholder to the next by pressing the Tab key. The placeholder character □ can be entered either from the Complete Characters palette (look under Letter-like Forms and then Keyboard Forms), or directly from the keyboard by typing ESC-sp-ESC (pressing the Escape key, then the characters `s` and then `p`, and finally the closing Escape key).

```
In[3]:= ButtonBox["Integrate[□,{□,□,□}]", Active->True]
        //DisplayForm
```

```
Out[3]//DisplayForm=
```

Integrate[□, {□, □, □}]

Clicking on this button causes the following expression to be pasted. You can move from one placeholder to another by pressing the Tab key.

```
Integrate[□, {□, □, □}]
```

ButtonStyle

Although having buttons that paste their contents at the current selection point is useful, there is much more that buttons can do. For example, they can wrap the contents of the `ButtonBox` around a selected expression and then evaluate that expression. To change the default behavior of buttons from simply pasting their contents to other actions, we have to use the `ButtonStyle` option. `ButtonStyle` is used to control both the style and the actions associated with your buttons. In the following example, `ButtonStyle` is set to `CopyEvaluateCell`.

```
In[4]:= ButtonBox["Integrate[■,x]", Active → True,
          ButtonStyle → "CopyEvaluateCell"] // DisplayForm
```

The ■ character is entered either from palettes or directly from the keyboard by typing ESC-spl-ESC. Evaluating the above input produces the cell below. Selecting the input cell containing `Cos[x^2] + x^5` and then clicking the button causes the template to be wrapped around the selected expression and then it is evaluated.

```
Out[4]//DisplayForm=
Integrate[■, x]

In[5]:= Cos[x^2] + x^5

In[6]:= Integrate[Cos[x^2] + x^5, x]

Out[6]= x^6/6 + Sqrt[π/2] FresnelC[Sqrt[2/π] x]
```

If you were to use `ButtonStyle→EvaluateCell` instead of `CopyEvaluateCell`, the button action would erase the selection and replace it with the new input and the result.

Another very useful `ButtonStyle` is `Hyperlink`. Making a hyperlink is accomplished by creating a button out of some expression and setting the `ButtonStyle` option to `Hyperlink` and adding the `ButtonData` option.

```
Cell[TextData[{
  "Search for button on ",
  ButtonBox["Google",
    ButtonData:>{
      URL[ "http://www.google.com"], None},
    ButtonStyle->"Hyperlink"]
}], "Text"]
```

The formatted version of this cell looks like this:

Search for button on Google

Setting `ButtonStyle` to `Hyperlink` sets the button action to jump to some location. That location is specified as the value of the option `ButtonData`. In this example, that is set to `URL["http://www.google.com"]`. `ButtonData` set to a URL will cause your web browser to be launched and opened to the location given as the argument to URL – in this case http://www.google.com.

A list of all the possible `ButtonStyle` values is displayed in Table 10.1.

ButtonStyle values	**Action**
`Paste`	pastes the contents (default)
`Evaluate`	pastes, then evaluates in place
`EvaluateCell`	paste, then evaluate entire cell
`CopyEvaluate`	copy current selection into new cell, then paste and evaluate
`CopyEvaluateCell`	copy current selection into new cell, then paste and evaluate cell
`Hyperlink`	jump to different location

Table 10.1: Possible `ButtonStyle`s and associated actions

ButtonFunction

Whenever you need to put some *Mathematica* code inside your button, you will need to do so as the value of the option `ButtonFunction`. You will also need to explicitly set the option `ButtonEvaluator` which is set to `None` by default. The `ButtonEvaluator` option tells the front end what program it should communicate with to process the contents of the button function. Setting it to `None` tells the front end to communicate with itself which is fine for operations like copying and pasting. But for operations that need to communicate with a kernel, you will have to specify that explicitly. A value of `Automatic` sends the code to the default kernel for the current notebook. If you had other kernels set up, you could direct the button function at one of those.

```
In[7]:= ButtonBox["Compute 5!",
          Active → True,
          ButtonFunction :→ Factorial[5],
          ButtonEvaluator → Automatic] // DisplayForm
```

Out[7]//DisplayForm=

Compute 5!

Clicking this button will not cause any output to be displayed. This is because these buttons are not evaluated in the kernel in the usual way as part of the main loop. In this case, you can use `Print` to see the side effect of this computation.

```
In[8]:= ButtonBox["Compute 5!",
          Active → True,
          ButtonFunction :→ Print[Factorial[5]],
          ButtonEvaluator → Automatic] // DisplayForm
```

Out[8]//DisplayForm=

Compute 5!

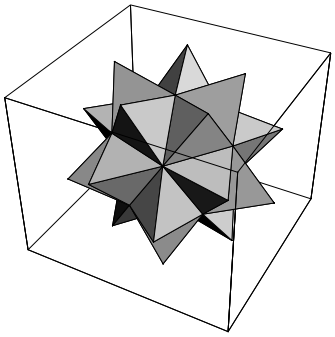
```
120
```

You can use any *Mathematica* function you wish as the value of the `ButtonFunction` option. But, in addition to the above issue with displaying output, you should be aware of another important issue. As it turns out, the front end does not know how to parse the special shorthand notation we often use for arithmetic and other operations. You will be forced to use the `FullForm` of such expressions inside of your `ButtonFunction`. So instead of `2+2`, use `Plus[2,2]`; instead of `{<<Graphics`;LogPlot[Exp[x],{x,1,2}]}` use `CompoundExpression[Get["Graphics`",LogPlot[Exp[x],{x,1,2}]]`. Fortunately, the parser for the front end can recognize the shorthand notation for `List`, `Rule`, and `RuleDelayed`, so you can use the shorthand notations { }, →, and :→, respectively.

As a final example, we will create a button that loads a package and then performs a computation with some functions from that package. Here is the code that we want to encapsulate in our button.

```
In[9]:= Needs["Graphics`Polyhedra`"]
```

In[10]:=
```
Show[Graphics3D[Stellate[Icosahedron[]]]];
```

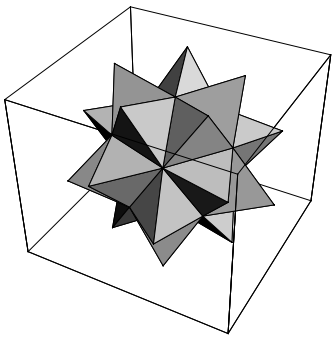


Here is the button code. Note that we have also added an option to `ButtonBox` to set the background and set the entire cell to use the Times font family.

```
Cell[BoxData[
    ButtonBox[RowBox[{"Stellate"," ","Icosahedron"}],
      ButtonFunction->
        CompoundExpression[Needs["Graphics`Polyhedra`"],
          Show[Graphics3D[Stellate[Icosahedron[]]]]],
      ButtonEvaluator->Automatic,
      Background->GrayLevel[.5]]],
      "Input",Active->True,
      FontFamily->"Times",
      FontColor->GrayLevel[1]]
```

And here is the button with a result of clicking it just below.

In[11]:= **Stellate Icosahedron**



Example: an evaluate button

At the end of Section 10.2 we created a function `EvaluateNext`, which evaluated the immediately following input cell. In this section we will turn the code from that function into a button.

Here was the code we developed in that section.

```
EvaluateNext:=(
    SelectionMove[EvaluationNotebook[],All,Cell];
    SelectionMove[EvaluationNotebook[],Next,Cell];
    SelectionEvaluate[EvaluationNotebook[]];
    )
```

To put this code inside a button, we need to make a few modifications. First, remember that the front end does not know how to parse shorthand notation such as `;`. Instead we need to use `CompoundExpression`. Second, instead of `EvaluationNotebook`, we will use `ButtonNotebook`, which gives the notebook in which the current button lives. Finally, we need to use `ButtonCell` to refer to the cell containing the button itself. Putting all these pieces together, here is the `ButtonFunction`.

```
In[13]:= ButtonFunction:→ CompoundExpression[
           {SelectionMove[ButtonNotebook[], All, ButtonCell],
            SelectionMove[ButtonNotebook[], Next, Cell],
            SelectionEvaluate[ButtonNotebook[]]}];
```

Here then is the code to generate our evaluate button.

```
Cell[TextData[{
  Cell[BoxData[
      ButtonBox["EVALUATE",
        ButtonFunction:>CompoundExpression[ {
            SelectionMove[
              ButtonNotebook[ ], All, ButtonCell],
            SelectionMove[
              ButtonNotebook[ ], Next, Cell],
            SelectionEvaluate[
              ButtonNotebook[ ]]}],
        Active->True]]],
  " MATHEMATICA INPUT"
}], "Text"]
```

And here is the formatted button. Clicking the Evaluate button causes the cell just below the button cell to be evaluated.

EVALUATE MATHEMATICA INPUT

```
In[14]:= 2 + 2

Out[14]= 4
```

Finally, let us add some formatting to make this cell a little nicer looking.

```
Cell[TextData[{
  Cell[BoxData[
      ButtonBox[
        StyleBox["EVALUATE",
          FontFamily->"Helvetica",
          FontSize->10,
          FontWeight->"Bold"],
        ButtonFunction:>CompoundExpression[ {
            SelectionMove[
              ButtonNotebook[ ], All, ButtonCell],
            SelectionMove[
              ButtonNotebook[ ], Next, Cell],
            SelectionEvaluate[
              ButtonNotebook[ ]]}],
        Active->True,
        Background->GrayLevel[0.500008]]]],
  StyleBox[" MATHEMATICA INPUT",
    FontFamily->"Helvetica",
    FontSize->10,
    FontWeight->"Bold",
    FontSlant->"Italic",
    FontColor->GrayLevel[1]]
}], "Text",
  Background->GrayLevel[0.500008]]
```

Here is the formatted version of this code with the result of clicking the button.

EVALUATE ***MATHEMATICA INPUT***

```
In[15]:= 2 + 5

Out[15]= 7
```

There is a little inefficiency in our code as we are calling the kernel several times (two instances of `SelectionMove` and one of `SelectionEvaluate`) for what are essen-

tially front end operations, moving and selecting. You can send these sorts of commands directly to the front end by wrapping them in `FrontEndExecute`. To distinguish between the kernel command and the front end command you also need to append the `FrontEnd`` context to the function. So for example, instead of using `SelectionMove[...]` in the kernel, you can send it directly to the front end with the following.

```
FrontEndExecute[FrontEnd`SelectionMove[…]]
```

With this in mind, the EVALUATE button can be rewritten by only changing the `ButtonFunction`.

```
ButtonFunction:>FrontEndExecute[ {
    FrontEnd`SelectionMove[
        ButtonNotebook[ ], All, ButtonCell],
    FrontEnd`SelectionMove[
        ButtonNotebook[ ], Next, Cell],
    FrontEnd`SelectionEvaluate[
        ButtonNotebook[ ]]}]
```

Another method of directly accessing front end commands is via front end tokens. These tokens allow you to perform any menu command directly from the kernel. We will not discuss them here, but for a detailed discussion of front end tokens, see the Front End category of the Help Browser.

Exercises

1. Create a button that will serve as a template for the `Plot3D` function.
2. Create a button that will wrap `Expand[]` around any selected expression and evaluate that expression.
3. Using `GridBox`, create a palette of buttons that operate on polynomials like that in Exercise 2. Include in your palette a button for each of `Expand`, `Factor`, `Apart`, and `Together`.