

# Using Built-In Functions

## 4

### What Are Functions?

Built-in functions are commands provided by Access and VBA that return a value. The value returned is dependent on the purpose of the function and the arguments, if any, passed to it. VBA is rich in functions that perform a variety of tasks and calculations for you. There are functions to convert data types, perform calculations on dates, perform simple to complex mathematics, make financial calculations, manage text strings, format values, and retrieve data from tables, among others.

Functions return a value, and most accept arguments to act on. A lot of your code will use functions to make calculations and manipulate data. You should familiarize yourself with the functions that VBA makes available to you, but don't expect to memorize their syntax. Between Intellisense and the VBA Help screens you can't go far off course, especially because Intellisense prompts you for each argument. If you need help understanding an argument, press F1 or look up the function in VBA Help.

Although this book was not meant to be a reference for VBA functions, this chapter explains many of the most used ones to give you an idea of VBA's power.

A point to remember when coding your functions: Be consistent in using data types. If you provide arguments of the wrong data type or assign a function to a different data type, you will cause an error.

### IN THIS CHAPTER

What Are Functions? .....	37
Converting Data Types .....	38
Working with Date Functions .....	42
Using Mathematical Functions .....	48
Using Financial Functions .....	50
Manipulating Text Strings .....	52
Formatting Values .....	55
Domain Aggregate Functions .....	59
Using the Is Functions .....	61
Interaction .....	61
Case Study: Add Work Days .....	64

## TIP

I don't know of any developer who knows every available function off the top of his or her head, so don't expect or think you need to. The more you code, the more you will remember, so feel free to use the references Microsoft provides. Use the Help option from the VBE menu to open the Developer Reference. In the search box type **functions list**, and one of the options is Functions (Alphabetical List). This gets you to a listing of all functions. Most of the function names are meaningful, so it shouldn't be difficult to find a function for the task you have.

## NOTE

In this chapter we frequently use the term *expression*. In my use an expression can be as simple as a value or text string or as complex as a formula using multiple operators and functions. Just remember that an expression expresses a value.

## Converting Data Types

At times you might find the need to import or link to data from external sources, or you might have to use data differently than the planned purpose. In such cases, the need may arise for you to convert from one data type to another. VBA includes several functions for this purpose. When you use a conversion function, the function returns the converted value but doesn't change the stored value.

→ For more on data types see, "VBA DataTypes" p. 28.

This chapter goes over some of the more commonly used conversion functions. You can find a full list by opening the Developers Reference using the VBE Help menu and searching on *type conversion functions*.

- **CBool**—Converts a value to a Boolean data type.
- **CDate**—Converts a value to a Date data type.
- **CInt**—Converts a value to an Integer data type.
- **CStr**—Converts a value to a String data type.
- **CVar**—Converts a value to a Variant data type.

## TIP

The most current conversion functions are prefixed with the letter *C*. It's better to use these functions in your conversions; however, you will also find included in VBA an older set of functions such as `Str` or `Val` for backward compatibility. The more current functions take your system's settings into account, whereas the older ones don't.

**TIP**

The `Val()` function has a use in addition to being a simple conversion function. It will return all numeric characters until it reaches a nonnumeric one. `CStr()` will return an error if you attempt to convert a string that contains nonnumeric data. For example, `Val("123abc")` will return the number 123 and `CInt("123abc")` will return a datatype mismatch error.

These functions have a simple syntax in common:

```
functionname(argument)
```

where *functionname* is the name of the function and *argument* is a value, variable, constant, or expression. The value of the argument is converted to a different data type depending on the function used, so it can be used elsewhere in your application. The value(s) used in the argument remain unchanged. It should be noted that not every data type can be converted to any other data type. The following sections explain the limitations.

## Converting to a Boolean Data Type

A Boolean value is either `True` or `False`. The `False` value is either the number or character zero (0). Any other value is considered `True`. If the argument passed to the `CBool` function evaluates to a zero, `CBool` returns a `False`. If it evaluates to any other value, `CBool` returns a `True`. For example; all the following return a `True` because the arguments all evaluate to a nonzero value:

```
CBool("1")
```

```
CBool(1+0)
```

```
CBool(5)
```

```
CBool(-50)
```

Conversely, the following expressions return a `False` because each argument evaluates to zero:

```
CBool(0)
```

```
CBool("0")
```

```
CBool(15-15)
```

The argument passed to the `CBool` function must contain all numeric characters or operators. If you use alphabetic characters you get a type mismatch error. One place where using `CBool` becomes useful is in conditional statements. For example, you might need to determine whether two values match. In our Inventory application you might need to determine whether you are out of stock on an item. You could use the following expression, which would return a `False` if the incomings matched the outgoings:

```
CBool(Sum(Incoming) - Sum(Outgoing))
```

## Converting to a Date Data Type

The `CDate` function converts any valid date/time value to a Date/Time data type. A valid date/time value can be either a number or a string that is formatted as a date or time. `CDate` determines valid date/time formats according to the regional settings you have chosen in Windows. You can use the following points to understand how dates are converted by `CDate`:

- If the argument is a numerical value, `CDate` converts the integer portion of the number according to the number of days since December 30, 1899. If the argument contains a decimal value, it's converted to a time by multiplying the decimal by 24 (for example, .25 would be 6:00 a.m.).
- If the argument is a string value, `CDate` converts the string if it represents a valid date. For example; "1/16/51", "March 16, 1952", and "6 Jun 84" would all be converted to a date. However, "19740304" would result in a type mismatch error.
- Access recognizes dates from January 1, 100, to December 31, 9999. Dates outside that range result in an error.
- I recommend that you use four-digit years for clarity. However, Access will work with two-digit years. If you enter a year less than 30, Access assumes you want a date in the twenty-first century. If you use a year of 30 or higher, it is assumed to be a twentieth century date.
- Remember that the `/` is also the division operator and the `-` is used for subtraction. So, if you enter dates such as 12/3/04 you will get unexpected results. Entering `CDate(12/3/04)` returns December 31, 1899, because 12 divided by 3 divided by 4 = 1. So you need to put such dates within quotes.

## Converting to an Integer Data Type

The `CInt` function takes a numeric or string value and converts it to an Integer data type. The *argument* is required and needs to represent a value within the range of -32,678 to 32,767. If the argument contains a decimal, Access rounds to the next whole number. A value of .5 or higher is rounded up; anything lower is rounded down. Some examples of `CInt` functions follow:

`CInt(10.5) = 11`

`CInt(25.333) = 25`

`CInt(10/3) = 3`

`CInt("1,000") = 1000`

TIP

That last example illustrates one of the advantages of `CInt` over the older `Val` function. `CInt` uses the system's regional settings and, therefore, recognizes the thousands separator, whereas `Val` would convert "1,000" to 1.

The argument must evaluate to a numeric value; otherwise, it returns an error. If the argument evaluates to a value outside the range of the Integer data type, you get an overflow error.

## Converting to a String Data Type

The `CStr` function converts just about every numeric value into a String data type. The required argument can be any variable, constant, expression, or literal value that evaluates to a string.

### CAUTION

If you use a variable as the argument, make sure it's been initialized to a value. If you use `CStr` on an uninitialized variable, it returns a numeric value of 0.

## Converting to a Variant Data Type

As I mentioned in the discussion of VBA data types in Chapter 3, "Using Variables, Constants, and Data Types," the `Variant` data type is the most flexible because it can accept almost any value. With `CVar`, you can convert just about any numeric or text string to the `Variant` data type. With numeric values there is a constraint to the same range for the `Double` data type.

### CAUTION

`CVar` should be used only when there is a doubt of the data type you are converting or when the data type isn't important.

## Converting Null Values

If you try to use a Null value in many expressions, you will probably encounter an error. For example, the following expression results in a runtime error if either of the values contains a Null:

```
varTotal = ValueA * ValueB
```

To avoid such errors you can utilize the `Nz` function to convert the value to a non-Null. The `Nz` function uses the following syntax:

```
Nz(value, [valueifnull])
```

The `Nz` function works similarly to an Immediate If (`IIF`) function. The following expressions are functionally equivalent:

```
varTotal = IIF(IsNull(ValueA),0,ValueA) * IIF(IsNull(ValueB),0,ValueB)
```

```
varTotal = Nz(ValueA,0) * Nz(ValueB,0)
```

The `valueifnull` is an optional argument; it defaults to 0 or a zero-length string based on the value's data type.

## Working with Date Functions

VBA has many functions that help you deal with dates. As long as you understand how Access stores Date/Time values, you should have no problem in working with date functions and values.

→ For a description of the Date/Time datatype see “VBA DataTypes,” p. 28.

In this section we go over most of the functions you use when dealing with dates.

### Returning the Current Date

To return the current date (as stored on your system) use the following function, which gives you a number counting the days from 12/30/1899:

```
Date()
```

How this value is displayed depends on your regional settings. You can use the `Date$()` function to return a 10-character string representing the date. This string uses the format *mm-dd-yyyy*. The `Date()` function returns only the system date; if you need to include the time use the `Now()` function. As noted earlier, a date/time value is a number where the integer portion represents the date and the decimal portion represents the time. So the `Now()` function will return an integer and decimal that represents the current date and time. The `Now()` function defaults to displaying its value according to the regional settings on your PC. On my PC it displays 7/25/2007 5:06:34 PM.

### Performing Date Arithmetic

Because dates are stored as numbers, you can do date arithmetic simply by adding or subtracting date values. However, VBA gives you a better way, the `DateAdd` function. Using this function, you can add 14 days, 14 weeks, 14 months, or 14 years to any date. Or you can find a time 60 hours earlier than the specified date and time.

The following is the syntax for `DateAdd`, where *interval* is a string that indicates the type of time period that you want to calculate:

```
DateAdd(interval, value, date)
```

Table 4.1 shows the various strings that can be entered as intervals. The *number* argument is a value or expression that specifies the number of intervals you want to calculate. The number used is an integer. If a decimal value is included, it's rounded to the nearest whole number, before performing the calculation. The *date* argument is a Date/Time value that is the base value to use in the calculation.

**Table 4.1 Interval Settings**

String Setting	Description
yyyy	Years
q	Quarters
m	Months
y	Day of year
d	Days
w	Weekdays
ww	Weeks
h	Hours
n	Minutes
s	Seconds

The *y*, *d*, and *w* intervals work interchangeably in the `DateAdd` function but have more meaning in other Date/Time functions. If the interval evaluates to a negative number, it returns an earlier date/time; a positive number returns a future date/time.

## Determining the Difference Between Two Dates

The `DateDiff` function is used to determine the number of intervals between two date/time values. The following is the syntax for the `DateDiff` function, where *interval* is a string that indicates the type of time period used to calculate the difference between the first and second dates represented by *date1* and *date2* (refer to Table 4.1):

```
DateDiff(interval, date1, date2[, firstdayofweek[, firstweekofyear]])
```

Also included in the `DateDiff` function are two optional arguments: *firstdayofweek* and *firstdayofyear*. These are numerical constants that can be used to adjust the first day of a week or year when using the `DateDiff` function. Tables 4.2 and 4.3 show a list of the values for each constant. The default values are Sunday and January 1, respectively.

**Table 4.2 First Day of Week Constants**

Constant	Description	Integer Value
vbSunday	Sunday (the default)	1
vbMonday	Monday	2
vbTuesday	Tuesday	3
vbWednesday	Wednesday	4
vbThursday	Thursday	5
vbFriday	Friday	6
vbSaturday	Saturday	7

**Table 4.3** First Week of Year Constants

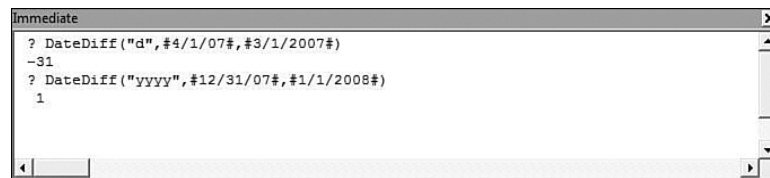
Constant	Description	Integer Value
vbFirstJan1	Use the week in which January 1 occurs (the default).	1
vbFirstFourDays	Use the first week that has at least four days in the new year.	2
vbFirstFullWeek	Use the first full week of the new year.	3

The results from this function might not always be as expected:

- If *date2* falls before *date1*, the function yields a negative value.
- The `DateDiff` function calculates a year has passed when a new year falls between the two dates, even if there are fewer than 365 days. So when using 12/31 and 1/1 as *date1* and *date2*, respectively, the function returns a 1.

Figure 4.1 shows how these guidelines affect the function in the Immediate window.

**Figure 4.1**  
The `DateDiff` function  
in action.



```

Immediate
? DateDiff("d", #4/1/07#, #3/1/2007#)
-31
? DateDiff("yyyy", #12/31/07#, #1/1/2008#)
1

```

NOTE

Notice that the dates in Figure 4.1 are enclosed by *octothorpes* (#—commonly known as a *pound sign*). This character is used to delimit date values, similarly to the way quotation marks are used with text strings. Access may recognize a date value and automatically insert the octothorpes, but it's a good practice to insert them yourself.

## Extracting Parts of Dates

The `DatePart` function is used to extract a portion of a date from a date value. A Date/Time data type contains several components that correspond to the intervals listed in Table 4.1. For example, the following expressions return the values 4, 1, and 2007, respectively:

```
DatePart("m", #4/1/2007#)
```

```
DatePart("d", #4/1/2007#)
```

```
DatePart("yyyy", #4/1/2007#)
```

The `DatePart` function uses the following syntax, where *interval* is a String value that defines the part of the date you want to extract and *date* is a valid Date/Time value (refer to Table 4.1 for a list of interval values):

```
DatePart(interval, date[, firstdayofweek[, firstweekofyear]])
```



Also included in the `DatePart` function are two optional arguments: *firstdayofweek* and *firstdayofyear*. These are numerical constants that can be used to adjust the first day of a week or year when using the `DatePart` function. Tables 4.2 and 4.3 show a list of the values for each constant. The default values are Sunday and January 1, respectively.

**TIP** Because you can extract any portion of a Date/Time value, it makes the most sense to store a date or time once as a valid Date/Time value. For example, even if you need to show only the month and year for a date, it would make sense to store a full date even if it's just the first or last day of the month.

## Creating Dates from the Individual Parts

With `DatePart` you extract part of a date; conversely, with the `DateSerial` function you combine the parts of a date to return a date value. The `DateSerial` function uses the following syntax, where *Year*, *Month*, and *Day* can be any expression that evaluates to an integer value that represents the respective date part:

```
DateSerial(Year, Month, Day)
```

There are some rules for each of the arguments:

- *Year* is required and must be equal to an integer from 100 to 9999.
- *Month* is required, and integers from 1 to 12 (positive or negative) are considered.
- *Day* is required, and integers from 0 to 31 (positive or negative) are considered.

The `DateSerial` function can take integer values outside those ranges and calculate the difference to return a date value. This makes it very powerful if you use expressions for the arguments. For example, the following expression returns June 5, 2008 because the 18th month from the start of 2007 is June:

```
DateSerial(2007,18,5)
```

Similarly, the following returns May 15, 2007, by using the 30 days in April and adding the difference of 15 days to the next month:

```
DateSerial(2007,4,45)
```

Although this shouldn't be used as a substitute for `DateAdd` or `DateDiff`, it can make it easy to create dates from calculated values.

**TIP** The expression `DateSerial(2007,5,0)` returns 4/30/07. Using 0 for the *Day* value can then be used to get the last day of a month. If you use `DateSerial(Year,Month+1,0)` you get the last day of the *Year* and *Month* used as arguments passed to the function.

## Creating Dates from String Values

The `DateValue` function can be used to return a date value from a string value; it uses the following syntax, where *stringexpression* must conform to the formats used by the system's Regional settings:

`DateValue(stringexpression)`

The following three expressions return the date June 1, 2007:

`DateValue("6/1/2007")`

`DateValue("June 1, 2007")`

`DateValue("1 Jun 07")`

**TIP** The functions `TimeSerial` and `TimeValue` perform similarly to the `DateSerial` and `DateValue` functions with time values.

## Extracting a Specific Date or Time Portion

Table 4.4 lists several functions that return a specific portion of a date or time value. The syntax for these functions is simple:

`Functionname(date/time)`

**Table 4.4 Date Component Functions**

Function	Result
<code>Day(date)</code>	Returns the day of the month as an integer between 1 and 31
<code>Hour(time)</code>	Returns the hour as an integer between 0 and 23
<code>Minute(time)</code>	Returns the minute as an integer between 0 and 59
<code>Second(time)</code>	Returns the second as an integer between 0 and 59
<code>Month(date)</code>	Returns the month as an integer between 1 and 12
<code>Year(date)</code>	Returns the year as an integer between 100 and 9999

## A Conversion and Date Example

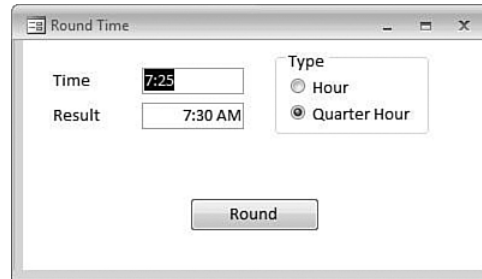
Sometimes you might need to round a time value to the nearest quarter hour or hour. This example uses some of the conversion and date/time functions previously discussed to accomplish that task.

1. Create a blank form and put two text boxes on it. Label the boxes `txtTime` and `txtResult`.
2. Add an option group to the form with the options Hour and Quarter Hour. Name the group `optType`.
3. Add a button to the form (turn off the wizard first). Name the button `cmdRound`.
4. Set the Record Selectors and Navigation buttons to No. Set Scroll Bars to neither.
5. In the On Click event of the button use the following code:

```
Private Sub cmdRound_Click()  
Dim intHrs As Integer, intMin As Integer  
Dim dteTime As Date  
' convert entered time to Time value  
  
dteTime = CDate(Me.txtTime)  
'extract parts of time  
  
intHrs = DatePart("h", dteTime)  
intMin = DatePart("n", dteTime)  
  
If Me.optType = 1 Then 'test for nearest type  
    'Round to nearest hour  
    If intMin >= 30 Then  
        dteTime = DateAdd("h", 1, dteTime)  
        dteTime = DateAdd("n", -intMin, dteTime)  
    Else  
        dteTime = DateAdd("n", -intMin, dteTime)  
    End If  
Else  
    'Round to quarter hour  
    Select Case intMin  
        Case Is < 8  
            intMin = 0  
        Case 8 To 23  
            intMin = 15  
        Case 24 To 38  
            intMin = 30  
        Case 39 To 53  
            intMin = 45  
        Case Else  
            intHrs = intHrs + 1  
            intMin = 0  
        End Select  
    dteTime = TimeSerial(intHrs, intMin, 0)  
End If  
  
'Populate Result control  
Me.txtResult = dteTime  
  
End Sub
```

6. Save form as `frmRound` (see Figure 4.2).

**Figure 4.2**  
The completed frmRound showing an example of input and result.



## Using Mathematical Functions

VBA provides a rich, broad set of functions to perform mathematical and financial calculations. There are too many to cover in this section, so we provide an overview of the most commonly used functions.

### The Abs Function

The Abs function returns the absolute value of a number, removing the sign. The following is the syntax for the Abs function, where *number* is any expression that evaluates to a numerical value:

`Abs (number)`

For example; this expression returns a 7:

`Abs (-7)`

### The Int Function

The Int function removes any decimal value from a number, returning the integer portion. The function uses the following syntax, where *number* is any expression that evaluates to a numerical value:

`Int (number)`

For example; this expression returns 15 because it truncates the value, removing the decimal portion:

`Int (15.9)`

However, if the numerical value is negative, Int returns the nearest negative integer, so the following returns -16:

`Int (-15.9)`

Although seemingly the same, Int and CInt can't be used interchangeably. The Int function doesn't convert the data type of the argument. Using CInt is often the better option, but it doesn't always return the same result. So be careful in determining which one to use.

## The Rnd Function

The `Rnd` function is used to generate a random number. It can be used with an optional argument represented as any valid numerical expression. The following is the syntax for the function:

`Rnd(seed)`

`seed` can be used to control the generated number as indicated in the following:

- If `seed` is a negative value, `Rnd` generates the same number.
- If `seed` is a positive number (other than 0) `Rnd` generates the next number in an internally determined sequence of numbers.
- If `seed` equals 0, `Rnd` generates the most recently generated number.
- If `seed` is omitted, `Rnd` generates the next number in an internally determined sequence of numbers.

The `Rnd` function generates a number in the range of 0 to 1, so if you need a whole number, you will have to multiply the generated value by a power of 10 and use the `Int` function to get your whole number.

**TIP** Use the `Randomize` statement to reset the internal sequence so that `Rnd` generates apparently unique values that are repeated.

4

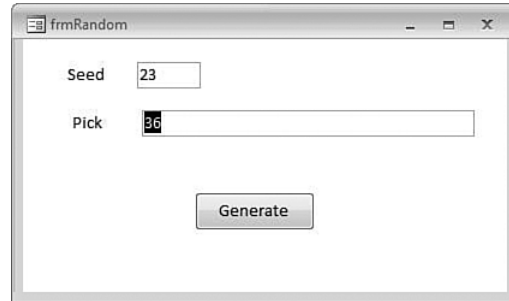
## A Mathematical Functions Example

To illustrate mathematical functions, let's create a function to generate a number between 1 and 99.

1. Create a blank form and put two text boxes on it. Label the boxes `txtSeed` and `txtPicks`.
2. Add a button to the form (turn off the wizard first). Name the button `cmdGenerate`.
3. Set Record Selectors and Navigation buttons to No. Set Scroll Bars to neither.
4. In the `OnClick` event of the button use the following code:

```
Private Sub cmdGenerate_Click()  
'Generate number between 1 and 99  
Me.txtPicks = Int(Rnd(Me.txtSeed) * 100)  
End Sub
```
5. Save form as `frmGenerate` (see Figure 4.3).

**Figure 4.3**  
The completed `frmGenerate` showing an example of a generated number.



## Using Financial Functions

Financial functions are used to perform many standard financial calculations such as interest rates, annuity or loan payments, and depreciation. Following are some financial functions you might find useful.

### The Ddb Function

The `Ddb` function calculates the depreciation of an asset for a specified time using the predefined double-declining balance method. The following is the syntax for this function, where *cost* is an expression representing the asset's opening cost and *salvage* is an expression that specifies the value of the asset at the end of *life*, an expression representing the term of the asset's lifespan.

```
Ddb(cost, salvage, life, period [, factor])
```

The *period* argument represents the time span for which the depreciation is being calculated. All these arguments use `Double` data types. There is an optional *factor* argument that specifies the rate of decline of the asset. If omitted, the double-declining method is used.

### The FV Function

The `FV` function is used to calculate the future value of an annuity. The `FV` function returns a double data type and uses the syntax

```
FV(rate, nper, pmt [, pv [, type]])
```

where *rate* is an expression resulting in a `Double` data type that represents the interest rate per period, *nper* is an expression resulting in an `Integer` data type that represents the number of payment periods in the annuity, and *pmt* is an expression resulting in a `Double` value that specifies the payment being made for each period. There are two optional arguments, *pv* and *type*, which are `Variant` data types that specify the present value of the annuity and whether payments are made at the start or end of each period.

## The Pmt Function

The `Pmt` function is used to calculate the payment for an annuity or loan. This function uses the syntax

```
Pmt(rate, nper, pv[, fv[, type]])
```

where *rate* is an expression resulting in a `Double` data type that represents the interest rate per period, *nper* is an integer expression that defines the number of payments to be made, and *pv* identifies the present value and is also a `Double` data type. There are two optional arguments, *fv* and *type*, which are `Variant` data types that represent the future value of the payments and whether payments are made at the start or end of each period.

## The Rate Function

The `Rate` function is used to calculate the periodic interest rate for an annuity or loan. The syntax for this function is

```
Rate(nper, pmt, pv[, fv[, type[, guess]])
```

Where *nper* is an expression resulting in a `Double` data type that represents the number of period, *pmt* is an expression resulting in a `Double` data type that represents the payment per period, and *pv* is an expression resulting in a `Double` data type that defines the present value. There are also three optional arguments: *fv*, *type*, and *guess*, which identify the future value, determine whether payments are made at the start or end of each period, and allow you to give an estimate of the rate, respectively.

## A Financial Functions Example

In keeping with the Inventory application theme of the sample file, this example looks at a scenario where you want to expand to cover a new product line. Because this new product line is from a new vendor, the vendor requires you to make a significant purchase the first time around. You don't have the \$10,000 to make the initial purchase, so you need to figure out different loan scenarios to see whether you can afford a loan.

1. Open the `basUDFs` module or one of your own.

2. Enter the following procedure:

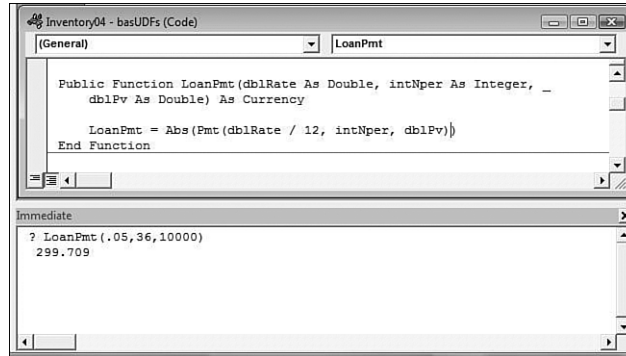
```
Public function LoanPmt(dblRate As Double, intNper As Integer, _
    dblPv As Double) As Currency
    LoanPmt = Abs(Pmt(dblRate/12, intNper, dblPv))
End Function
```

3. In the Immediate window enter the following statement and press Enter:

```
? LoanPmt(.05,36,10000)
```

Figure 4.4 shows the code and the result. This loan would cost you \$300 per month for 36 months. You can now try out different scenarios with combinations of rate and term to see what your payments might be.

**Figure 4.4**  
The LoanPmt function  
and its results.



## Manipulating Text Strings

You use string functions to manipulate groups of text data. The following are some examples of where you might use string functions:

- Checking to see whether a string contained another string
- Parsing out a portion of a string
- Replacing parts of a string with another value

The following string functions help you do all these tasks and more.

### The Asc Function

Every individual character can be represented by a number value. These values are listed in the American Standard Code for Information Interchange (ASCII). To return the ASCII value of a character use the following syntax, where *string* is an expression that results in a Text data type. It returns an integer value between 0 and 255.

`Asc(string)`

#### CAUTION

The Asc function reads only the first character of the string if there are multiple characters.

With any text string you must use apostrophes or quotation marks to define and delineate the text string; otherwise, Asc returns an error. However, if the string is a numeric, the delimiters can be eliminated. For example, the following two functions both return the value 51:

`Asc("3")`

`Asc(3)`



## The Chr Function

The `Chr` function is the reverse of the `Asc` function. Whereas `Asc` returns the numerical value from character, `Chr` returns the character from a number. The following is the syntax for this function, where *charactercode* is an integer value between 0 and 255:

```
Chr(charactercode)
```

As you saw previously, the character 3 is represented by the number 51. So the following functions returns a 3:

```
Chr(51)
```

NOTE

The numbers 0–255 represent the values of characters according to the ASCII table. An example of that table can be found at <http://www.asciitable.com>.

TIP

Many of the string functions return a value as a variant of the `String` subtype. An alternative set of string functions add a `$` to the function name (for example, `Chr$`). These alternative functions return a literal string value. This provides a better performance because VBA doesn't have to evaluate the data type during processing.

## The Case Functions

There is actually no case function. There are two functions, `LCase` and `UCase`, that can be used to change the case of a text string. They use the following syntax, where *string* is an expression that returns a string value. Both functions return the *string* in either lowercase or uppercase, respectively.

```
LCase(string)
```

```
UCase(string)
```

TIP

You can use the `UCase` function to convert entered data so that the data entry person doesn't have to concern himself with entering the proper case.

## The Len Function

The `Len` function is used to determine the number of characters in a text string. This function uses the following syntax, where *string* is an expression that results in a `Text` data type. The function returns a long integer except where the string is `Null`, in which case it returns a `Null` value.

```
Len(string)
```

## The Left, Right, and Mid Functions

Among the most used functions, these three return a portion of a string depending on the function and the arguments provided. All three result in a `Variant Long` subtype but support a `$` version, which forces a `String` data type.

The `Left` and `Right` functions use a similar syntax:

```
Left(string, length)
```

```
Right(string, length)
```

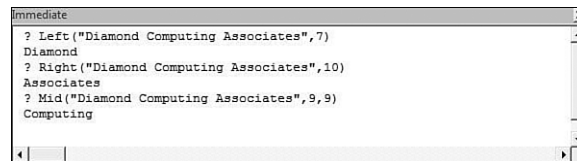
Here, *string* is an expression that results in a `Text` data type to be parsed and *length* is an expression that results in an `Integer` data type that specifies the number of characters from either the left or right end of the string to return.

The `Mid` function can parse a text string from any part of the string. It uses the following syntax, where *string* is a required argument that represents an expression resulting in a `Text` data type and *start* is a required argument that specifies where to start parsing the string:

```
Mid(string, start[, length])
```

An optional argument, *length*, specifies how many characters from the *start* point to parse. If *length* is omitted or is greater than the number of characters to the end of the string, all characters from *start* are returned. Figure 4.5 shows the three functions parsing various parts of the same string.

**Figure 4.5**  
The `Left`, `Right`, and `Mid` functions parsing the same text.



## The Replace Function

The `Replace` function is used to replace one or more characters within a string with a different character or characters. This function takes the following syntax, where *string* is an expression representing the text string to be searched, *stringtoreplace* is an expression representing the string to be searched for, and *replacementstring* represents the string you want in place of *stringtoreplace*

```
Replace(string, stringtoreplace, replacementstring[, start[, count[, compare]])
```

In addition, there are three optional arguments: *start*, which specifies where to start searching within the string; *count*, which specifies the number of replacements to process; and *compare*, which is a constant indicating the method used to compare *stringtoreplace* with *string*. Table 4.5 lists the constants that can be used.

**Table 4.5 Comparison Constants**

Constant	Value	Description
vbUseCompareOption	-1	Performs a comparison using the setting of the Option Compare statement.
vbBinaryCompare	0	Performs a binary comparison.
vbTextCompare	1	Performs a textual comparison.
vbDatabaseCompare	2	Microsoft Access only. Performs a comparison based on information in your database.

## The Split Function

The `Split` function takes a delimited string and populates an array with the parts. The following is the syntax for the `Split` function, where *string* is a delimited string of values:

```
Split(string[, delimiter[, count[, compare]])
```

This is the only required argument. The first optional argument is *delimiter*, which specifies the delimiting character separating the values. If you omit *delimiter* a space is assumed to be the *delimiter*. The second optional argument is *count*, which limits the number of values parsed. For example, there might be five values separated by commas in the string, but a *count* argument of 3 parses only the first three values. The final optional argument is *compare*. See Table 4.5 for the comparison constants.

## The Trim Functions

Three functions can be used to trim leading or trailing spaces from a string. All three use the same syntax, where *string* is an expression that results in a `Text` data type:

```
Trim(string)
```

```
LTrim(string)
```

```
RTrim(string)
```

The `Trim` function removes both leading and trailing spaces, `LTrim` removes the leading spaces, and `RTrim` removes the trailing spaces. All three functions return a `Variant String` subtype and support the `$` format to force a `Text` data type.

## Formatting Values

Often data is stored differently from the way it's displayed on forms and in reports. The `Format` function is your tool to change how data is displayed. Access provides many predefined formats for you to use and allows you to customize your own formats. For example, a phone number might be stored as 10 digits but you can display it like (111) 222-3333 by applying a format. Another example are `Date/Time` values. As previously noted, they are stored as a `Double` number. However, the `Format` function can display the number in a variety of date or time formats.

**CAUTION**

Keep in mind that the `Format` function returns a `Variant String` subtype, which will probably be different from the original value's data type, and that the original data remains unchanged. This means that you should use `Format` only for display purposes; you don't want to use it in calculations.

The `Format` function uses the following syntax, where *expression* can be either a `String` or `Numeric` data type that results in the value you want to format:

```
Format(expression[, format[, firstdayofweek[, firstweekofyear]])
```

There are three optional arguments, the first of which determines how the data is formatted. The other two optional arguments, *firstdayofweek* and *firstdayofyear*, are numerical constants that can be used to adjust the first day of a week or year when using the `DatePart` function. Tables 4.2 and 4.3 show a list of the values for each constant. The default values are Sunday and January 1, respectively.

Tables 4.6 and 4.7 show some of the predefined formats you can use.

**Table 4.6 Numeric Named Formats**

Format	Example	Result
General Number	<code>Format(12345.6789, "General Number")</code>	12345.6789
Currency	<code>Format(12345.6789, "Currency")</code>	\$12,345.68
Fixed	<code>Format(0.1, "Fixed")</code>	0.10
Standard	<code>Format(12345.6789, "Standard")</code>	12,345.68
Percent	<code>Format(6789, "Percent")</code>	67.89%
Scientific	<code>Format(12345.6789, "Scientific")</code>	1.23E+03
Yes/No	<code>Format(0, "Yes/No")</code> <code>Format(3, "Yes/No")</code>	No Yes
True/False	<code>Format(0, "Yes/No")</code> <code>Format(3, "Yes/No")</code>	False True
On/Off	<code>Format(0, "Yes/No")</code> <code>Format(3, "Yes/No")</code>	Off On

The result for `Currency` is based on the United States regional settings; if you use a different regional setting, the `Currency` format uses those settings. For the `Boolean` types a zero results in a `No`, `False`, or `Off` result. Any other value gives the opposite result.

**Table 4.7 Date/Time Named Formats**

Format	Example	Result
General Date	Format("04/01/07", "General Date")	4/1/2007
Long Date	Format("04/01/07", "Long Date")	Sunday April 1, 2007
Medium Date	Format("04/01/07", "Medium Date")	01-Apr-07
Short Date	Format("04/01/07", "Short Date")	4/1/2007
Long Time	Format('13:13:13', "Long Time")	1:13:13 PM
Medium Time	Format('13:13:13', "Medium Time")	1:13 PM
Short Time	Format('13:13:13', "Short Time")	13:13

## Applying User-Defined Formats

Although the predefined formats listed in Tables 4.6 and 4.7 cover many situations, at times you'll need to create your own formats. You can use a number of special characters and placeholders to define your own formats. Tables 4.8, 4.9, and 4.10 list these formats.

**Table 4.8 Numeric User-Defined Formats**

Format	Explanation	Example	Result
0	Display actual digit or 0 for each 0 used. Rounds if more digits than shown.	Format(12.3456, "000.00000") Format(12.3456, "000.00")	012.34560 012.35
#	Display actual digit or nothing. Rounds if more digits than shown.	Format(12.3456, "###.#####") Format(12.3456, "###.##")	12.3456 12.35
%	Multiples by 100 and adds percent sign	Format(.3456, "##%")	35%
E- E+ e- e+	Display scientific notation.	Format(1.234567, "###E-###")	123E-2
- + \$ ()	Display a literal character.	Format(123.45, "\$###.##")	\$123.45
\	Display following character as a literal.	Format(.3456, "##.##\%")	.35%

**Table 4.9 Date User-Defined Formats**

Format	Explanation	Example	Result
d	Display day of month without leading zero	Format("04/04/07", "d")	1
dd	Display day of month with leading zero where needed	Format("04/04/07", "dd")	01
ddd	Display abbreviated day of week	Format("04/01/07", "ddd")	Sun
dddd	Display full day of week	Format("04/01/07", "dddd")	Sunday
ddddd	Display short date	Format("04/01/07", "ddddd")	4/1/2007
dddddd	Display long date	Format("04/01/07", "dddddd")	Sunday, April 1, 2007
m	Display month without leading zero	Format("04/01/07", "m")	4
mm	Display month with leading zero	Format("04/01/07", "mm")	04
mmm	Display abbreviated month name	Format("04/01/07", "mmm")	Apr
mmmm	Display full month name	Format("04/01/07", "mmmm")	April
q	Display quarter of year	Format("04/01/07", "q")	2
h	Display hours without leading zero	Format("13:13:13", "h")	1
hh	Display hours with leading zero	Format("13:13:13", "hh")	01
n	Display minutes without leading zero	Format("13:07:13", "n")	7
nn	Display minutes with leading zero	Format("13:07:13", "nn")	07
s	Display seconds without leading zero	Format("13:13:07", "s")	7
ss	Display seconds with leading zero	Format("13:13:07", "ss")	07
tttt	Display 12-hour clock	Format("13:13:13", "tttt")	1:13:13 PM
AM/PM	With other time formats displays either upper- or lowercase AM/PM	Format("13:13:13", "hh:nn AM/PM")	1:13 PM
am/pm		Format("13:13:13", "hh:nn am/pm")	1:13 pm
A/P	With other time formats displays either upper- or lowercase A/P	Format("13:13:13", "hh:nn A/P")	1:13 P
a/p		Format("13:13:13", "hh:nn a/p")	1:13 p
ww	Display the number of the week (1–54)	Format("04/01/07", "ww")	14

Format	Explanation	Example	Result
w	Display the number of the day of the week	Format("04/01/07", "w")	1
y	Display the day of the year (1–366)	Format("04/01/07", "y")	91
yy	Display 2-digit year (00–99)	Format("04/01/07", "yy")	07
yyyy	Display 4-digit year (0100–9999)	Format("04/01/07", "yyyy")	2007

These formats can also be combined to display different date or time formats. The following are some examples:

Format("04/01/07", "yyyymmdd") = 20070401

**TIP** This format is useful when exporting data to other formats and still maintaining chronological sort.

Format("4/01/07", "mmm dd") = Apr 01

Format("04/01/07", "mmm yyyy") = Apr 2007

**Table 4.10 String User-Defined Formats**

Format	Explanation	Example	Result
@	Display actual character or space	Format("VBA", "@@@@")	VBA
&	Display actual character or nothing	Format("VBA", "&&&&")	VBA
<	Display character as lowercase	Format("VBA", "<<<<")	vba
>	Display character in uppercase	Format("VBA", ">>>>")	VBA

## Domain Aggregate Functions

Domain Aggregate functions are specific to Microsoft Access because they are used to retrieve data from tables. Because you can't assign the results of a query directly to a variable, you must use Domain Aggregate functions to retrieve that data. There are other ways besides Domain Aggregate functions that will be covered later in this book. The advantages of the Domain Aggregate functions are that they can accept a set of criteria to retrieve just the data needed. All the Domain Aggregate functions use a similar syntax, where *expression* is the name of the field in a table or query, *domain* is the name of the table or query, and *criteria* is a comparison to define which record to extract the value from:

Function("[*expression*]", "*domain*", *criteria*)

Notice that the *expression* is usually surrounded by quotes and brackets and that the *domain* is also surrounded by quotes. I'll list some of the more commonly used Domain Aggregate functions.

### The DLookup Function

The DLookup function is used to retrieve a value from a single field in a table or query. The following example returns the last name of the contact for Supplier G from tblSuppliers in the Inventory application:

```
DLookup("[LastName]", "tblSuppliers", "[Company] = ' & "Supplier G" & "'")
```

The DLookup function retrieves that value from the first record matching the criteria. Because *Company* is a Text data type, you must concatenate the single quotes around the company name. If you are comparing a Numeric data type, no quotes are needed, and a Date/Time data type requires octothorpes (#) to delimit the value to be searched for.

### The DCount Function

The DCount function is used to count the number of records in a table or query that match your criteria. An example of the DCount function follows:

```
DCount("*", "tblEmployees", "[Jobtitle] = 3")
```

This returns a result of 6 because there are six employees whose job title is Sales Representative.

### The DMax/DMin Functions

The DMax and DMin functions return the highest or lowest values in the domain according to the criteria listed. An example of the DMin function follows:

```
DMin("[CreatedDate]", "tblTransactions")
```

This returns 3/22/2006 4:02:28 PM, which is the earliest transaction in the Transactions table.

#### TIP

The DMax function is often used to produce a sequential numbering system that can be dependent on some other value. For example, say you wanted to number each transaction for each employee and start the numbering each time a new employee is added. In such a case you could use the following code snippet in the After Update event of the control where the employee is selected on your form:

```
Me.txtIncrement =  
Nz(DMax("[Increment]", "tblTransactions", "[EmployeeID] = " &  
Me.cboEmployee), 0) + 1
```

This sets the control named txtIncrement to the highest value of the field Increment plus 1 for the selected employee. If no record for the employee is found, the NZ function causes the expression to return a 0, which is then incremented to 1.



## Using the Is Functions

VBA provides a series of functions to help you trap errors that might arise from data type mismatches. These functions test a value to see whether it's a specified type.

- **IsArray**—Checks whether the value is an array
  - **IsDate**—Checks whether the value is a Date/Time data type
  - **IsEmpty**—Checks whether the value hasn't been initialized with a value
  - **IsError**—Checks whether an expression results in an error
  - **IsMissing**—Checks whether an optional argument has been passed to a procedure
  - **IsNull**—Checks whether the value contains a Null
  - **IsNumeric**—Checks whether the value is a number
  - **isObject**—Checks whether a variable contains a reference to an object
- We cover arrays in more detail in Chapter 7, "Working with Arrays."

All these functions use the same syntax, where *value* is a value or expression being tested:

```
IsFunction(value)
```

The functions all return a Boolean data type, either `True` if the value meets the condition being checked or `False` if it doesn't.

## Interaction

At times you need to provide information to the application's user or get information from them. This is interacting with the users. Two functions that can perform such an action are the `MsgBox` and `InputBox` functions.

### The `MsgBox` Function

You use the `MsgBox` function to present information to users with an opportunity to respond to the information. You have control over how the message box appears and what response the user can make. The `MsgBox` function uses the following syntax, where *prompt* is the only required argument and represents a text string that constitutes the message presented by the message box:

```
MsgBox(prompt[, buttons][, title][, helpfile, context])
```

The users can respond through a choice of one or more buttons. Table 4.11 lists various button options you can use. You can supply a string value for *title* that displays in the title bar of the message box. The other two optional arguments—*helpfile* and *context*—are seldom used and go together. The *helpfile* argument is a string that points to a help file to be used if the user clicks the message box's Help button. The *context* argument is a numeric value that specifies a number to be used within the help file. (Note: Creating help files is outside the scope of this book.)

**Table 4.11** MsgBox Button Constants

Constant	Description	Integer Value
vbOkOnly	OK button	0
vbOkCancel	OK and Cancel buttons	1
vbAbortRetryIgnore	Abort, Retry, and Ignore buttons	2
vbYesNoCancel	Yes, No, and Cancel buttons	3
vbYesNo	Yes and No buttons	4
vbRetryCancel	Retry and Cancel buttons	5

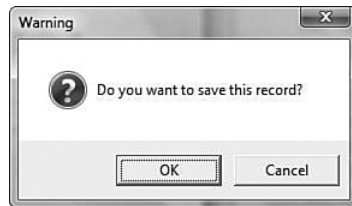
Table 4.12 lists constants for the icons that can be displayed in the message box. You can display both icons and buttons using the following syntax:

buttonconstant + iconconstant

As an example, the following function displays the message box shown in Figure 4.6. There are two buttons—OK and Cancel—and a question mark icon.

```
MsgBox("Do you want to save this record?", vbOkCancel + vbQuestion, "Warning")
```

**Figure 4.6**  
A message box asking whether the user wants to save a record.

**CAUTION**

Besides the MsgBox function there is also a MsgBox action. The action displays the MsgBox without returning a value as a response.

**Table 4.12** Icon Constants

Constant	Description	Integer Value
vbCritical	Critical message	16
vbQuestion	Warning message	32
vbExclamation	Warning message	48
vbInformation	Information message	64

When the user clicks one of the buttons, the function returns its value. Table 4.13 shows the values returned for each button.

**Table 4.13 Button Values**

Button	Returned Value	Integer Value
OK	vbOK	1
Cancel	vbCancel	2
Abort	vbAbort	3
Retry	vbRetry	4
Ignore	vbIgnore	5
Yes	vbYes	6
No	vbNo	7

The following code snippet is built around the message box function previously shown:

```
Private Function cmdSave_OnClick()
Dim strMsg As String
strMsg = "Do you want to save this record?"
If MsgBox("strMsg, vbOKCancel + vbQuestion, "Warning") = vbOK Then
    DoCmd.RunCommand acCmdSaveRecord
Else
    Me.Undo
End If
```

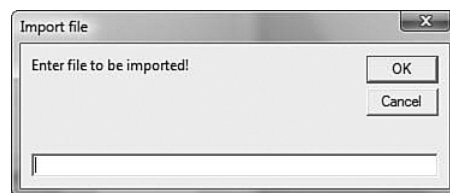
## The InputBox Function

The Inputbox function displays a dialog box with a prompt that allows the user to enter a value that can then be assigned to a variable (see Figure 4.7). The following is the syntax for this function, where *prompt* is a String that displays a message for the user and is the only required argument:

```
InputBox(prompt[, title][, default][, xpos][, ypos][, helpfile, context])
```

The message is used to let the user know what data should be input. The *title* is a String that is displayed in the title bar of the window. The *default* is used to set a default value when the box opens. The *xpos* and *ypos* arguments allow you to precisely position the box in terms of the top and left of the screen. The *helpfile* and *context* arguments are the same as for the MsgBox.

**Figure 4.7**  
An input box asking the user to enter a filename.



You usually use `InputBox` to retrieve a value from the user during processing of code. An example follows:

```
Dim strFilename As String
strFilename = InputBox("Enter file to be imported!", "Import file")
DoCmd.TransferText acExportDelim, , "Import", strFilename
```

**TIP**

I rarely use the `InputBox` function, preferring to use a form to allow the user to supply input. Using a form gives you much greater control over the input. With a form you can use interactive controls such as combo boxes or option groups to ensure that the correct data is entered. We'll deal with this more in later chapters.

## CASE STUDY

### Case Study: Add Work Days

Sometimes you might need to figure a delivery or follow-up date for a shipment. You want to calculate such dates based on business days, not calendar days. The following function allows you to enter a start date and the number of business days and returns the date equal to that number of business days:

1. Open the UDFs module or a new one.
2. Enter the following code into the module:
 

```
Public Function AddWorkdays(dteStart As Date, intnNumDays As Integer) As Date

    Dim dteCurrDate As Date
    Dim i As Integer

    dteCurrDate = dteStart
    AddWorkdays = dteStart
    i = 1

    Do While i < intnNumDays
        If Weekday(dteCurrDate, vbMonday) <= 5 AND _
            IsNull(DLookup("[Holiday]", "tblHolidays", "[HolDate] = #" & _
                dteCurrDate & "#")) Then
            i = i + 1
        End If

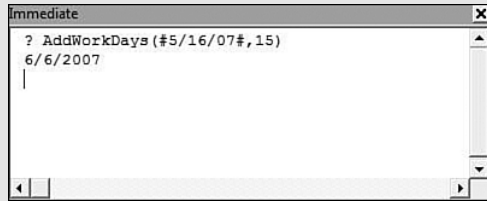
        dteCurrDate = dteCurrDate + 1

    Loop
    AddWorkdays = dteCurrDate

Exit_AddWorkDays:
End Function
```
3. Test the code by entering the following into the Immediate window. Figure 4.8 shows the results.
 

```
? AddWorkDays(#5/16/07#, 15)
```

**Figure 4.8**  
The results from using the  
AddWorkDays function.



```
immediate
? AddWorkDays(#5/16/07#,15)
6/6/2007
```