2 Mächtige benutzerdefinierte Funktionen entwickeln

Jeder Excel-Anwender, der über das Anfängerstadium hinaus ist, kennt den Funktionsassistenten. Über den Funktionsassistenten stehen dem Benutzer hunderte von nützlichen Funktionen zur Verfügung. Genug, sollte man meinen. Benutzt man aber Excel häufig und auch für sehr spezielle Anwendungen, so kommt man früher oder später in eine Situation, in der selbst dieses breite Funktionsangebot nicht ausreicht.

Die Lücke kann man mit benutzerdefinierten Funktionen (englisch: user-defined functions bzw. UDF, manchmal auch custom functions genannt) schließen. Dies sind selbstprogrammierte Funktionen, die im Funktionsassistenten angeboten werden, zusätzlich zu den standardmäßig in Excel verfügbaren Funktionen (den sog. Worksheet Functions).

2.1 Ideale VBA-Funktionen

Natürlich müssen alle Funktionen, die wir im Rahmen dieses Kapitels betrachten oder schreiben, den Regeln von VBA zur Erstellung von Funktionen genügen. Doch die Einhaltung dieser Regeln alleine genügt nicht, wenn wir wirklich gute Software schreiben wollen, also Software, die den Qualitätszielen aus dem ersten Kapitel genügt.

Deshalb sollen in diesem Abschnitt einige Eigenschaften herausgearbeitet werden, welche ideale Funktionen unbedingt besitzen sollten. Wir studieren hierzu ein Beispiel: eine Funktion, welche einer Zahl ihr Quadrat zuordnet. Zunächst betrachten wir diese Funktion in ihrer mathematischen Form, denn unsere programmierten Funktionen orientieren sich am mathematischen Begriff der Funktion:

$$y = x^2$$

Es ist üblich, eine solche Funktionsgleichung durch Angaben des Definitionsbereichs (mögliche Werte von *x*) und des Bildbereichs (mögliche Werte von *y*) zu ergänzen, damit Benutzer der Funktion wissen, in welchen Zusammenhängen sie angewandt werden kann. Hier ist der Definitionsbereich die Menge der reellen Zahlen, und auch der Bildbereich entspricht dieser Menge.

Eine adäquate VBA-Fassung dieser Funktion ist

```
Public Function Quadrieren (ByVal x As Double) As Double

    Quadrieren = x ^2

End Function
```

Beachten Sie folgende Eigenschaften dieser Formulierung der Funktion:

- Die unabhängige Variable *x* ist als Variable in der Parameterklammer der Kopfzeile aufgeführt. Ihr Definitionsbereich (Datentyp Double) ist explizit genannt und mit der Genauigkeit beschrieben, die durch Angabe eines Datentyps möglich ist.

- Der Bildbereich ist ebenfalls explizit in der Kopfzeile beschrieben. Dies geschieht mit dem Passus„As Double“ nach der Parameterklammer.
- Im Rumpf der Funktion wird der Funktionswert aus der Variablen *x* ermittelt, die in der Parameterklammer genannt ist. In diese Berechnung geht kein weiterer Input ein. Die Ermittlung des Funktionsergebnisses ist außerdem das Einzige, was die Funktion bewirkt.

Die Betrachtung lässt sich leicht auf Funktionen mit mehreren unabhängigen Variablen erweitern. Auch für solche Funktionen lassen sich VBA-Funktionen schreiben, welche außer den in der Parameterklammer aufgeführten Variablen keinen weiteren Input verarbeiten und deren Effekt einzig die Ermittlung des Funktionsergebnisses ist.

Von allen Funktionen, die wir im Rahmen dieses Kapitels erstellen, wollen wir, dass sie die oben beschriebenen Eigenschaften haben. Funktionen dieser Art sind ideal im Sinne der Ziele der Softwareentwicklung. Sie sind in sich geschlossene Einheiten mit wohldefiniertem Input und Output, die problemlos in andere Programme verpflanzt werden können und einfach zu warten sind.

2.2 Funktionen und benutzerdefinierte Funktionen

In diesem Abschnitt betrachten wir VBA-Funktionen unter mehr formalen Gesichtspunkten als im vorhergegangenen: Welche Regeln gelten für das Verfassen von Funktionen und welche gelten für die so genannten benutzerdefinierten Funktionen, denen unser besonderes Augenmerk in diesem Kapitel gilt.

Die im vorangegangenen Abschnitt postulierten idealen Funktionseigenschaften sind davon unabhängig zu sehen. Weder garantiert das Einhalten der VBA-Regeln für benutzerdefinierte Funktionen eine ideale Funktion im obigen Sinn, noch ist jede im obigen Sinne ideale Funktion als benutzerdefinierte zugelassen.

Eine **Funktion in VBA** ist ein durch Kopf- und Fußzeile eingeschlossenes Teil eines Moduls mit folgenden Eigenschaften

- Eine Funktion kann nicht für sich alleine laufen, sondern wird aufgerufen. Der **Aufruf** kann aus einer Prozedur heraus erfolgen oder, wenn die Funktion eine UDF ist, von einem Tabellenblatt aus. Die aufrufende Stelle können wir **Auftraggeber** nennen.
- Eine Funktion liefert ein **Ergebnis** an ihren Auftraggeber. Dabei kann es sich um einen Einzelwert (z.B. eine Zahl) handeln, um eine Kollektion von Zahlen (wie z.B. ein Zahlenarray), um einen benutzerdefinierten Datentyp oder um ein Objekt (in der objektorientierten Programmierung). Das gelieferte Ergebnis entspricht dem Bildelement (der abhängigen Variablen) beim mathematischen Begriff der Funktion.
- Eine Funktion kann keine oder mehrere **Parameter** (Argumente) haben. Die Parameter entsprechen den Originalelementen (unabhängigen Variablen) beim mathematischen Begriff der Funktion. Der Fall mit null Parametern ist ein Grenzfall, der aber gar nicht so selten ist. Eine Funktion ohne Parameter kann beispielsweise eine häufig benötigte Konstante oder einen zufällig ausgewählten Wert liefern.

- Funktionen können **Public** oder **Private** deklariert werden (in der objektorientierten Programmierung auch mit **Friend**). Die Deklaration Private bewirkt, dass die betreffende Funktion nur innerhalb ihres Moduls aufgerufen werden kann.
- Funktionen können auf Zellen in einem Excel-Tabellenblatt zugreifen und sie auch verändern. Beachten Sie aber: im Allgemeinen vermeidet man das Lesen von Zelleninhalten aus einer Funktion heraus; es ist schädlich in Bezug auf die Wartbarkeit und die Wiederverwendbarkeit und ist mit der im vorangegangenen Abschnitt beschriebenen idealen Form der Funktion nicht vereinbar.

Beispiel 1: eine Funktion, die einen Einzelwert liefert

Die folgende Funktion Zufallszahl produziert eine gleichverteilte Zufallszahl vom Typ Double im Bereich [xmin, xmax[, also einen numerischen Einzelwert. Sie macht Gebrauch von der eingebauten VBA-Funktion Rnd, welche eine gleichverteilte Zufallszahl im Intervall [0, 1[liefert

```
Public Function Zufallszahl (ByVal xmin As Double, ByVal xmax As Double) As Double
    Zufallszahl = xmin + (xmax – xmin) * Rnd
End Function
```

Beispiel 2: eine Funktion, die ein Array liefert

Eine Funktion, die ein Array liefert, ist die unten stehende Funktion diff. Sie hat als Argument ein eindimensionales Array a von Double-Zahlen und ermittelt daraus die Differenzenfolge erster Ordnung.

```
Public Function diff (ByRef a() As Double) As Double()
    Dim d() As Double
    ReDim d(LBound(a) to UBound(a) -1)
    Dim i As Integer
    For i = LBound(d) to UBound(d)
        d(i) = a(i + 1) – a(i)
    Next i
    diff = d
End Function
```

Beispiel 3: eine Funktion, die einen benutzerdefinierten Datentyp liefert

Eine Funktion kann immer nur ein Ergebnis liefern. Es gibt jedoch Situationen, in denen diese Beschränkung hinderlich ist. Nehmen wir an, wir benötigen zwei Ergebnisse, die aus denselben Grunddaten ermittelt werden. Wir könnten für beide Ergebnisse jeweils eine Funktion schreiben und die beiden Funktionen nacheinander aufrufen. Dieses Vorgehen würde funktio-nieren, wäre aber nur die zweitbeste Lösung, wenn man die beiden Ergebnisse auch zugleich ermitteln könnte. Effizienter ist es in einem solchen Fall, einen benutzerdefinierten Datentyp zu bilden, der die beiden Ergebnisse zu einem bündelt. Hier ein Beispiel:

Wir möchten eine Funktion haben, die uns ein aus n Zufallszahlen zusammengestelltes Array sowie den Maximalwert daraus liefert. Hierfür deklarieren wir zunächst einen benutzerdefinierten Datentyp, den wir ZZ nennen, und der sich aus einem Double-Array r und einem einzelnen Double-Wert maxW zusammensetzt.

```
Type ZZ
        r() AS Double
        maxW As Double
End Type
```

Die Funktion ZufallZZ liefert eine Ausprägung des Datentyps ZZ. Dies funktioniert allerdings nur, wenn die Deklaration des Datentyps ZZ im Zugriff ist.

```
Public Function ZufallZZ (ByVal n As Integer) As ZZ
        Dim z As ZZ
        ReDim z.r(n – 1)
        z.maxW = 0
        Dim i As Integer
        For i = 0 to n - 1
                z.r(i) = Rnd
                If z.r(i) > z.maxW Then z.maxW = z.r(i)
        Next i
        ZufallZZ = z
End Function
```

Beachten Sie, dass die Zusammenstellung der Zufallszahlen und die Ermittlung des maximalen Werts innerhalb einer einzigen Schleife geschehen. Dies wäre nicht möglich gewesen, wenn wir die beiden Vorgänge auf zwei Funktionen verteilt hätten.

Benutzerdefinierte Funktionen

Von einer **benutzerdefinierten Funktion** (User Defined Function, UDF) spricht man, wenn eine in VBA programmierte Funktion auch aus Tabellenblättern heraus nutzbar ist. Generell wird jede mit Public deklarierte und ohne Syntaxfehler programmierte Funktion, die sich in einem **Standard**modul befindet, im Funktionsassistenten unter der Rubrik **benutzerdefiniert** angezeigt und ist damit im Prinzip als UDF nutzbar.

Damit diese Funktionen auch ohne Fehler als UDF genutzt werden können, sind jedoch einige Einschränkungen zu beachten:

- Bei den Parametern der Funktion sind nicht alle Datentypen möglich, die innerhalb von VBA zulässig sind. So führen benutzerdefinierte Datentypen bei den Parametern einer UDF zu Fehlermeldungen. Auch ein Array wird nicht als Parameter akzeptiert (sieht man davon ab, dass ein einfacher Variant-Parameter möglich ist, hinter dem sich dann ein Array verstecken kann).
- Der Datentyp des Funktionsergebnisses ist ebenfalls Einschränkungen unterworfen. Dies sind größtenteils dieselben wie für die Parameter, aber es gibt auch Unterschiede. Bei-

spielsweise wird eine Funktion, welche ein Range liefert (entspricht einem Bereich eines Tabellenblatts) nicht akzeptiert, wohl aber eine Funktion, welche einen Parameter vom Typ Range hat.

- Eine weitere Einschränkung besagt, dass benutzerdefinierte Funktionen nicht die Zelleninhalte von Tabellenblättern verändern dürfen. Das bloße Lesen von Zelleninhalten ist gestattet (aber nicht ratsam!).
- Die Namen benutzerdefinierter Funktionen sollten eindeutig sein. Auch darf man nicht einen Namen wählen, der bereits für eine der eingebauten Excel- oder VBA-Funktionen vergeben ist.

Welche der oben gezeigten Beispielfunktionen sind angesichts dieser Einschränkungen als UDF verwendbar?

- Die Funktion Zufallszahl (s. oben) würde als UDF akzeptiert, wenn es im deutschen Excel nicht bereits eine eingebaute Excel-Funktion dieses Namens gäbe. Benennt man die Funktion um, z.B. in ZufallszahlS, so taugt sie als UDF.
- Die Funktion diff (s. oben) kann, weil sie ein Double-Array als Input verlangt, nicht als UDF ausgeführt werden.
- Die Funktion ZufallZZ wird ebenfalls **nicht** als UDF akzeptiert, weil sie einen benutzerdefinierten Datentyp liefert.

2.3 Parameter und Funktionsergebnis exakt deklarieren

VBA erlaubt einen recht lockeren Umgang mit Datentypen. Dies kann zu überraschenden „Erfolgen“ führen. Betrachten wir die Funktion

```
Public Function f(x, y, z)
        f = x + y + z
End Function
```

Diese Funktion wird vom Übersetzer akzeptiert und sogar als benutzerdefinierte Funktion im Funktionsassistenten angezeigt, obwohl kein Datentyp für das Funktionsergebnis angegeben ist, und bei den Parametern weder ein Datentyp noch ein Übergabemodus (ByVal oder ByRef) gesetzt ist.

Man muss wissen, dass in einem solchen Fall, wo der Programmierer unvollständig deklariert, das VBA-System die Lücken mit Standardwerten füllt. Die Parameter und das Funktionsergebnis werden stillschweigend mit dem Datentyp Variant versehen, für den Übergabemodus wird ByRef ange-nommen. In unserem Beispiel führt das dazu, dass die Funktion f zu einer wahren „Allzweckfunktion“ wird. Wir können als Parameterwerte beispielsweise ganze Zahlen übergeben. Für die Werte x = 1, y = 2 und z = 3 liefert die Funktion dann das Ergebnis 6. Aber auch Wörter werden als Parameterwerte akzeptiert. Setzen wir für x, y und z die Werte “Max“, “und“ und “Moritz“, so ist das Funktionsergebnis die Zeichenfolge “MaxundMoritz“.

Eine Funktion wie f ist kaum auszutesten, weil man gar nicht weiß, was sie können soll und was nicht zulässig sein soll. Wegen ihrer Unberechenbarkeit ist sie eine potentielle Fehlerquelle in jedem Programm, in dem sie eingesetzt wird. Darüber hinaus verschwendet sie Speicherplatz, weil für

Daten vom Typ Variant mehr Platz benötigt wird als für andere Datentypen, wie z.B. Byte, Integer, Single oder Double.

Um Fehler und Ineffizienz zu vermeiden, sollte man stets die Datentypen für alle Parameter und den Funktionswert angeben. Eine sinnvolle Deklaration für die Funktion anstelle der obigen wäre z.B.

```
Public Function addieren (ByVal x As Double, ByVal y As Double, ByVal z As Double) As Double
```

2.4 Optionale Parameter

Funktionen mit optionalen Parametern sind sicherlich aus den eingebauten Excel-Funktionen bekannt. Wie man eine solche Funktion in VBA programmiert, zeigt das folgende Beispiel. Die Funktion liefert für eine Tagnummer von 1 bis 7 den Namen des betreffenden Wochentags. Normalerweise entspricht die Tagnummer 1 dem Montag, aber es existiert auch eine andere Zählweise, worin die Nummer 1 dem Sonntag entspricht. Unsere Funktion soll für beide Zählweisen die richtige Antwort geben. Dafür sorgt das optionale Argument beginn mit den möglichen Werten „sonntag“ und „montag“. In unserem Beispiel ist das optionale Argument beginn zusätzlich mit einem Standardwert (“montag“) versehen. Dieser Wert wird genommen, wenn die aufrufende Stelle den Parameter nicht mit einem Wert versorgt.

```
Public Function Wochentg4(ByVal tagnr As Integer, _
        Optional ByVal beginn As String = “montag”)  As String
    If LCase(beginn) = "sonntag" Then
       If tagnr = 1 Then tagnr = 7 Else tagnr = tagnr - 1
    End If

    If tagnr = 1 Then
       Wochentg4 = "Montag"
    ElseIf tagnr = 2 Then
       Wochentg4 = "Dienstag"
    ElseIf tagnr = 3 Then
       Wochentg4 = "Mittwoch"
    ElseIf tagnr = 4 Then
       Wochentg4 = "Donnerstag"
    ElseIf tagnr = 5 Then
       Wochentg4 = "Freitag"
    ElseIf tagnr = 6 Then
       Wochentg4 = "Samstag"
    Else
       Wochentg4 = "Sonntag"
    End If

End Function
```

2.5 Zerlegung und Delegation

Bei umfangreicheren Funktionen lohnt es sich, Teilaufgaben in separate Funktionen auszugliedern. Die Hauptfunktion enthält in diesem Fall Aufrufe der Unterfunktionen, sie delegiert Teilaufgaben. Zerlegung fördert die Lesbarkeit und Wartbarkeit der Anwendung. Die kleineren Einheiten lassen sich leichter verstehen und austesten.

In der folgenden Funktion sortiert, welche die Zeichen eines Strings aufsteigend sortiert, werden Teilaufgaben an die Funktionen minPos und entfernePos delegiert.

```
Public Function sortiert(ByVal s As String) As String
   Dim n As String
   n = ""
   Dim mp As Integer
   mp = 1
   Do While Len(s) > 0
     mp = minPos(s)
     n = n & Mid(s, mp, 1)
     s = entfernePos(s, mp)
   Loop
   sortiert = n
End Function
```

Um zu verhindern, dass alle aufgerufenen Funktionen ebenfalls als benutzerdefinierte Funktionen im Funktionsassistenten erscheinen, kann man diese mit Private deklarieren. Man muss sie aber in diesem Fall in dasselbe Modul einstellen wie die Oberfunktion

```
Private Function minPos(ByVal s As String) As Integer
   Dim mp As Integer
   mp = 1
   Dim i As Integer
   For i = 2 To Len(s)
     If Mid(s, i, 1) < Mid(s, mp, 1) Then mp = i
   Next i
   minPos = mp
End Function


Private Function entfernePos(ByVal s As String, ByVal pos As Integer) As String
   entfernePos = Left(s, pos - 1) & Right(s, Len(s) - pos)
End Function
```

Was sollen wir tun, wenn wir Unterfunktionen wie minPos und entfernePos auch von anderen Modulen aus benutzen wollen, wir aber nicht wollen, dass sie im Funktionsassistenten erscheinen?

Die Deklaration mit Private kommt hier nicht in Frage. Wir können solche Funktionen aber mit Public deklarieren und in ein separates Modul auslagern, dem wir die **Option Private Module** voranstellen. Dann können wir aus allen anderen Moduln darauf zugreifen; sie erscheinen aber nicht im Funktionsassistenten.

2.6 Aufruf von Funktionen mit benannten Parametern

Wenn eine Funktion viele Parameter hat, ist die Formulierung von Aufrufen ausgesprochen fehlerträchtig. Dies gilt noch mehr, wenn darunter optionale Parameter sind. Glücklicherweise gibt es in VBA die Möglichkeit, die Parameter beim Aufruf zu benennen. Man ist dann nicht mehr an eine bestimmte Reihenfolge gebunden.

Nehmen wir an, eine benutzerdefinierte Funktion BDF soll eine Teilaufgabe an die Funktion Beispielfunktion delegieren. Beispielfunktion hat die folgende Kopfzeile:

```
Private Function Beispielfunktion(ByVal Par1 As Double, _
                                  Optional ByVal Par2 As Integer, _
                                  Optional ByVal Par3 As Integer, _
                                  Optional ByVal Par4 As Double, _
                                  Optional ByVal Par5 As Integer) As Double
```

Nehmen wir außerdem an, von den optionalen Parametern werde in unserem Fall nur Par5 benötigt. Für den obligatorischen Parameter Par1 soll der aktuelle Wert einer Variablen v1 übergeben werden, für den optionalen Parameter Par5 der Wert 1. Der Aufruf in BDF könnte dann folgendermaßen lauten:

```
…
Dim d As Double
d = Beispielfunktion (Par1 := v1, Par5 := 1)
```

Beachten Sie, dass für die Zuordnung des Werts an den Parameternamen die Symbolkombination := benutzt werden muss, ein = genügt nicht.

Der Aufruf ohne namentliche Benennung der Parameter müsste mit zusätzlichen Kommata versehen sein, um die Auslassungen anzuzeigen. Auch müsste die Reihenfolge der Parameter aus der Kopfzeile der Funktion Beispielfunktion eingehalten werden. Hierbei können leicht Fehler auftreten.

```
…
Dim d As Double
d = Beispielfunktion (v1, , , , 1)
```

2.7 Benutzerdefinierte Funktionen, die Bereiche verarbeiten

Angenommen, wir benötigen eine Funktion für die Aufgabe, die Anzahl der positiven ungeraden Zahlen aus einem Bereich eines Tabellenblatts zu ermitteln. Um die Sache nicht unnötig kompliziert zu machen, wollen wir davon ausgehen, dass sich in dem Bereich nur ganze Zahlen befinden.

Wie bereits bemerkt, wird eine Funktion, welche ein Array als Parameter erwartet, nicht als benutzerdefinierte Funktion akzeptiert. Hätten wir also bereits eine Funktion

```
Public Function AnzahlPositivUngerade (ByRef a() As Long) As Long
```

so könnten wir sie nicht in dieser Form verwenden, auch wenn der zu untersuchende Tabellenbereich inhaltlich einem Array von ganzen Zahlen entspricht.

Für eine benutzerdefinierte Fassung der Funktion benötigen wir den Datentyp **Range**, der im sog. Objektmodell von Excel enthalten ist. Ein Range entspricht einem rechteckigen Bereich eines Tabellenblatts. Dies kann im Extremfall auch das gesamte Tabellenblatt sein. Die Funktion könnte wie folgt lauten:

```
Public Function AnzahlPositivUngerade(ByVal r As Range) As Long
  Dim a As Long
  a = 0
  Dim i As Long, j As Long
  For i = 1 To r.Rows.Count
     For j = 1 To r.Columns.Count
        If CLng(r(i, j).Value) Mod 2 <> 0 And CLng(r(i, j).Value) > 0 Then
           a = a + 1
        End If
     Next j
  Next i
  AnzahlPositivUngerade = a
End Function
```

Der Ausdruck r.Rows.Count bezeichnet die Anzahl der Zeilen in r, r.Columns.Count die Anzahl der Spalten. Diese Art der Formulierung (Punktnotation) ist im Abschnitt Programmierung mit dem Excel-Objektmodell näher erläutert.

Die obige Formulierung ist nicht die einzige mögliche. Verwenden wir den Schleifentyp mit For each, so können wir uns die Doppelschleife ersparen, und die Funktion wird noch etwas übersichtlicher:

```
Public Function AnzahlPositivUngerade (ByVal r As Range) As Long
  Dim a As Long
  a = 0
  Dim zelle As Range
  For Each zelle In r.Cells
      If CLng(zelle.Value) Mod 2 <> 0 And CLng(zelle.Value) > 0 Then
         a = a + 1
      End If
    Next
  AnzahlPositivUngerade = a
End Function
```

For each ist nur im Zusammenhang mit einer Collection anwendbar. Dies ist ein Datentyp aus der objektorientierten Programmierung. Eine Collection ist eine Sammlung von Objekten, in unserem Beispiel eine Sammlung von Zellen.

2.8 Funktionen durch Einwickeln in benutzerdefinierte Funktionen verwandeln

Funktionen, welche ein Range als Parameter entgegennehmen, lassen sich zwar als benutzerdefinierte Funktionen verwenden, sind aber für andere Anwendungen kaum zu gebrauchen. Im Gegensatz dazu sind Funktionen mit Array-Parametern zwar vielseitig einsetzbar, aber ausgerechnet nicht als benutzerdefinierte Funktionen. Soll man nun von allen Funktionen zwei Versionen bereit halten, oder gibt es eine dritte Möglichkeit?

Die gibt es in der Tat. Wir schreiben die Funktion zunächst in der nicht-benutzerdefinierten Form, also mit einem Array-Parameter ausgestattet. Dies liest sich wie folgt:

```
Public Function AnzahlPositivUngerade (ByRef a() As Long) As Long
  Dim i As Long, j As Long, anz As Long
  anz = 0
  For i = LBound(a, 1) To UBound(a, 1)
     For j = LBound(a, 2) To UBound(a, 2)
        If a(i, j) Mod 2 <> 0 And a(i, j) > 0 Then anz = anz + 1
     Next j
  Next i
  AnzahlPositivUngerade = anz
End Function
```

Danach schreiben wir eine benutzerdefinierte Version. Wir wollen Sie APU nennen. APU besteht nur aus einer Hülle, die um AnzahlPositivUngerade herum gelegt wird. Weil jedoch APU ein Range als Parameter verlangt, AnzahlPositivUngerade aber nur einen Array-Parameter akzeptiert, benötigen wir eine zusätzliche kleine Funktion RangeToLngArray, die ein Range in ein Long-Array überführt.

```
Public Function APU(ByVal r As Range) As Long
   APU = AnzahlPositivUngerade(RangeToLngArray(r))
End Function

Private Function RangeToLngArray(ByVal r As Range) As Long()
  Dim l() As Long
  ReDim l(1 To r.Rows.Count, 1 To r.Columns.Count)
  Dim i As Long, j As Long
  For i = 1 To UBound(l, 1)
     For j = 1 To UBound(l, 2)
        l(i, j) = CLng(r(i, j).Value)
     Next j
  Next i
  RangeToLngArray = l
End Function
```

Die Hüllenfunktion APU ist sehr kurz. Gewöhnlich genügt eine Zeile für den Rumpf einer solchen Funktion, die nur eine Hülle (englisch: wrapper) um eine andere Funktion bildet.

Der Aufwand für die Hilfsfunktion RangeToLngArray erscheint auf den ersten Blick groß, denn diese ist fast so lang wie die von Anfang an benutzerdefiniert programmierte Funktion AnzahlPositivUngerade. Allerdings muss man beachten, dass RangeToLngArray gut wieder verwendet werden kann. Diese Funktion kann in allen Fällen eingesetzt werden, wo eine benutzerdefinierte Funktion ein Range mit ganzen Zahlen verlangt, die ursprüngliche Version aber ein Long-Array verarbeitet. Auch spielt der Umfang der Aufgabenstellung eine Rolle. Wenn die zu schreibende Funktion mehrere Seiten umfasst, tritt der Aufwand für die Umwandlungsfunktion RangeToLngArray in den Hintergrund.

2.9 Benutzerdefinierte Funktionen, die Werte für Bereiche liefern

Unter den eingebauten Excel-Funktionen, die im Funktionsassistenten zur Verfügung stehen, befinden sich auch solche, die Werte für ganze Bereiche von Tabellenblättern liefern. Diese Funktionen sind überaus nützlich und werden häufig gebraucht, wie z.B. die Funktionen aus der Matrizenrechnung (MTRANS, MINV, MDET und MMULT).

Welchen Ergebnistyp soll eine UDF haben, die einen Bereich liefert? Auf den ersten Blick könnte man Range vermuten, weil dies der Datentyp des Excel-Objektmodells ist, der einem Bereich entspricht, und weil die Parameter einer solchen UDF zumindest teilweise von diesem Typ sind. Dass dies nicht gestattet ist, liegt daran, dass Range ein ganz besonderer Typ ist. Ein Range ist ein bestimmter Ausschnitt aus bestimmten Arbeitsblatt. Mit Hilfe einer Funktion könnte man daher keine neuen Range-Objekte erzeugen, sondern sich nur auf bestehende beziehen. Mit anderen Worten: das Ergebnis einer Funktion, welche ein Range liefert, würde immer an derselben Stelle ausgegeben. Die Schöpfer von VBA propagieren deshalb den Typ Variant für das Ergebnis von UDF, welche Werte für Bereiche liefern.

Im Hinblick auf die Prinzipien guter Programmierung ist aber auch Variant nicht die optimale Lösung für den Funktionstyp. Variant ist bekanntlich ein Allzwecktyp und führt häufig zu Fehlern im Code, außerdem vermindert seine Verwendung die Verarbeitungsgeschwindigkeit. Eine erheblich bessere Lösung für den Typ des Funktionsergebnisses ist ein Array eines konkreten Datentyps, also z.B. ein Array von Double oder Long. Erfreulicherweise wird zumindest in neueren Versionen von Excel-VBA auch ein solches Array als Ergebnistyp akzeptiert. Im Folgenden verwenden wir deshalb Arrays konkreter Datentypen (also nicht von Variant) als Ergebnistyp.

Wir betrachten ein einfaches Beispiel einer UDF, die ein Array liefert: die Multiplikation einer Matrix mit einem Skalar. Die Funktion MSkC hat zwei Parameter, einen Double-Parameter s für den Skalar s und einen Range-Parameter r für die Matrix.

```
Public Function MSkC(ByVal s As Double, ByVal r As Range) As Double()
  Dim a() As Double
  ReDim a(1 To r.Cells.Rows.Count, 1 To r.Cells.Columns.Count)
  Dim i As Integer, j As Integer
  For i = 1 To UBound(a, 1)
      For j = 1 To UBound(a, 2)
         a(i, j) = r(i, j) * s
      Next j
```

```
    Next i
    MSkC = a
End Function
```

Bei Funktionen wie MSkC taucht das Problem auf, das schon im Abschnitt 2.6 diskutiert wurde. Der Datentyp Range für den Parameter r ist an die spezielle Art der Eingabe über das Tabellenblatt gebunden; die Funktion taugt so nicht für universelle Verwendung. Es liegt nahe, auch hier die schon beschriebene Einwickeltechnik anzuwenden. Wir programmieren die Funktion zunächst als Funktion MSk, die ein Array entgegennimmt und ein Array liefert:

```
Public Function MSk(ByVal s As Double, ByRef a() As Double) As Double()
    Dim m() As Double
    ReDim m(1 To UBound(a, 1), 1 To UBound(a, 2))
    Dim i As Integer, j As Integer
    For i = 1 To UBound(m, 1)
        For j = 1 To UBound(m, 2)
            m(i, j) = s * a(i, j)
        Next j
    Next i
    MSk = m
End Function
```

Dann schreiben wir eine Hüllenfunktion, hier MSkCW genannt, mit der wir die Funktion MSk so einwickeln, dass sie als benutzerdefinierte Funktion akzeptiert wird. Zusätzlich brauchen wir noch eine Hilfsfunktion RangeToDblArray, welche die der benutzerdefinierten Funktion übergebenen Range-Objekte in die Double-Arrays überführt, die von MSk erwartet werden.

```
Public Function MSkCW(ByVal s As Double, ByVal rng As Range) As Double()
    MSkCW = MSk(s, RangeToDblArray(rng))
End Function


Public Function RangeToDblArray(ByVal r As Range) As Double()
    Dim d() As Double
    ReDim d(1 To r.Cells.Rows.Count, 1 To r.Cells.Columns.Count)
    Dim i As Integer, j As Integer
    For i = 1 To UBound(d, 1)
        For j = 1 To UBound(d, 2)
            d(i, j) = CDbl(r.Cells(i, j).Value)
        Next j
    Next i
    RangeToDblArray = d
End Function
```

2.10 Art und Zahl der Parameter offen halten mit ParamArray

Das Schlüsselwort ParamArray bezeichnet eine besondere Art von Funktionsparameter, die hohe Flexibilität bei der Gestaltung von benutzerdefinierten Funktionen erlaubt. Man kann ParamArray auch für Funktionen verwenden, welche nicht für den Einsatz als UDF gedacht sind, aber dafür ist es eigentlich nicht nötig.

ParamArray lässt sich am Besten anhand eines Beispiels erklären.

Nehmen wir an, dass wir eine Funktion schreiben möchten, welche aus einer Zahlentabelle eine oder mehrere Spalten entfernt. Genauer ausgedrückt: die Funktion soll eine Tabelle liefern, welche der Tabelle m entspricht, bis auf die besagten Spalten, die eben nicht enthalten sein sollen.

1	2	3	4
1	2	3	4
1	2	3	4

1	4
1	4
1	4

Das oben stehende Bild zeigt beispielhaft den Input und den Output einer solchen Funktion. Gegenüber der Input-Tabelle (links) wurden für das Funktionsergebnis (rechts) die beiden Spalten 2 und 3 entfernt.

Eine Funktion zu programmieren, die dies bewerkstelligt, wäre ein Leichtes, wenn wir wüssten, dass es stets die Spalten 2 und 3 sein sollen, die entfernt werden müssen. Aber dies können wir leider nicht voraussetzen. Es können beliebige und beliebig viele Spalten sein, die entfernt werden sollen, mit der einzigen Einschränkung, dass mindestens eine Spalte übrig bleiben muss.

Wenn wir zunächst vernachlässigen, dass die gewünschte Funktion als UDF verwendet werden soll, könnten wir die Funktion so konzipieren, dass wir ihr neben dem Input-Array m ein weiteres Array c übergeben, welches die Nummern der zu löschenden Spalten in aufsteigender Reihenfolge enthält. Eine solche Funktion könnte z.B. folgende Kopfzeile besitzen:

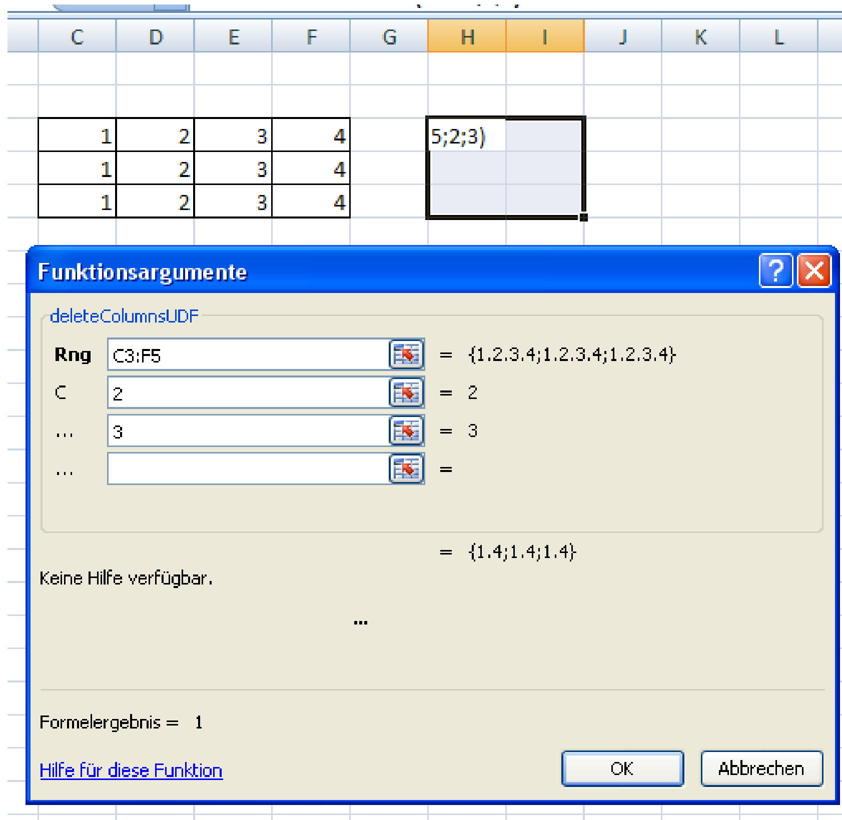
```
Public Function deleteColumns (ByRef m() As Double, ByRef c() As Integer) As Double()
```

Obwohl diese Funktion nicht UDF-tauglich ist, wollen wir doch ihre Programmierung betrachten, weil sie im Rahmen der späteren UDF-Lösung verwertet werden kann. In der folgenden Lösung delegiert die Hauptfunktion deleteColumns das eigentliche Löschen einzelner Spalten an die Funktion deleteColumn. Für jede zu löschende Spalte wird deleteColumn einmal aufgerufen.

```
Public Function deleteColumns(ByRef m() As Double, ByRef c() As Integer) As Double()
  Dim i As Long
  Dim n() As Double
  n = m
  For i = UBound(c) To 1 Step -1
     n = deleteColumn(n, c(i))
  Next i
  deleteColumns = n
End Function
```

```
Public Function deleteColumn(ByRef m() As Double, ByVal colno As Integer) As Double()
   Dim i As Long, j As Long, k As Long
   Dim n() As Double
   ReDim n(1 To UBound(m, 1), 1 To UBound(m, 2) - 1)
   For i = 1 To UBound(m, 1)
      k = 0
      For j = 1 To UBound(m, 2)
          If j <> colno Then
               k = k + 1
               n(i, k) = m(i, j)
          End If
      Next j
   Next i
   deleteColumn = n
End Function
```

An dieser Lösung ist nichts auszusetzen außer dass sie nicht als UDF einsetzbar ist. Bevor wir die UDF-taugliche Lösung diskutieren, soll kurz dargestellt werden, wie sie sich einem Excel-Benutzer präsentieren wird (folgendes Bild).



Der Benutzer hat in den Spalten H und I zunächst den Bereich selektiert, der das Ergebnis aufnehmen soll. Dann hat er mit Hilfe des Funktionsassistenten die Funktion deleteColumnsUDF aufgerufen. Im Fenster des Funktionsassistenten hat der Benutzer in der mit Rng bezeichneten Zeile bereits den Bereich angegeben, in dem sich die Input-Tabelle befindet. Darunter hat er die Spalten angegeben,

welche die Ergebnistabelle nicht enthalten soll, in diesem Fall die Spalten 2 und 3. Wie das Bild zeigt, könnte der Benutzer noch weitere auszusparende Spalten angeben. Dies hat er hier nicht getan. Stattdessen hat er die OK-Taste betätigt und hat dabei, weil es sich um eine Array-Funktion handelt, die Tastenkombination <Strg> + <^> gedrückt. Damit wird das Fenster geschlossen, und der markierte Bereich H3:I5 enthält das gewünschte Ergebnis.

Die hier benutzte UDF deleteColumnsUDF hat die Kopfzeile

```
Public Function deleteColumnsUDF(ByVal rng As Range,  _
                                 ParamArray c() As Variant) _
                        As Double()
```

Der erste Parameter rng ist der Bereich, in dem sich die Input-Tabelle befindet, der zweite Parameter enthält die auszusparenden Spalten. Der Zusatz **ParamArray** bewirkt das oben dargestellte, äußerst großzügige Eingabeverhalten des Assistenten: der Benutzer kann eine beliebige Anzahl von Spalten angeben.

In dem nun folgenden Code der Funktion wird die bereits bekannte **Einwickeltechnik** angewandt. Sie erlaubt, die bereits programmierten Funktionen deleteColumns und deleteColumn zu nutzen. Die zusätzlich herangezogene Funktion RangeToDblArray ist bereits aus einem früheren Abschnitt bekannt.

```
Public Function deleteColumnsUDF(ByVal rng As Range, _
                                 ParamArray c() As Variant)  _
                        As Double()
   Dim i As Integer, s() As Integer
   ReDim s(1 To UBound(c) + 1)     'weil c  0-basiert ist
   For i = 1 To UBound(s)
       s(i) = CInt(c(i - 1))
   Next i
   deleteColumnsUDF = deleteColumns(RangeToDblArray(rng), s)
End Function
```

Die Funktion tut nichts weiter, als den Input in eine Form zu bringen, der von der Funktion deleteColumns akzeptiert wird, und dann diese Funktion aufzurufen.

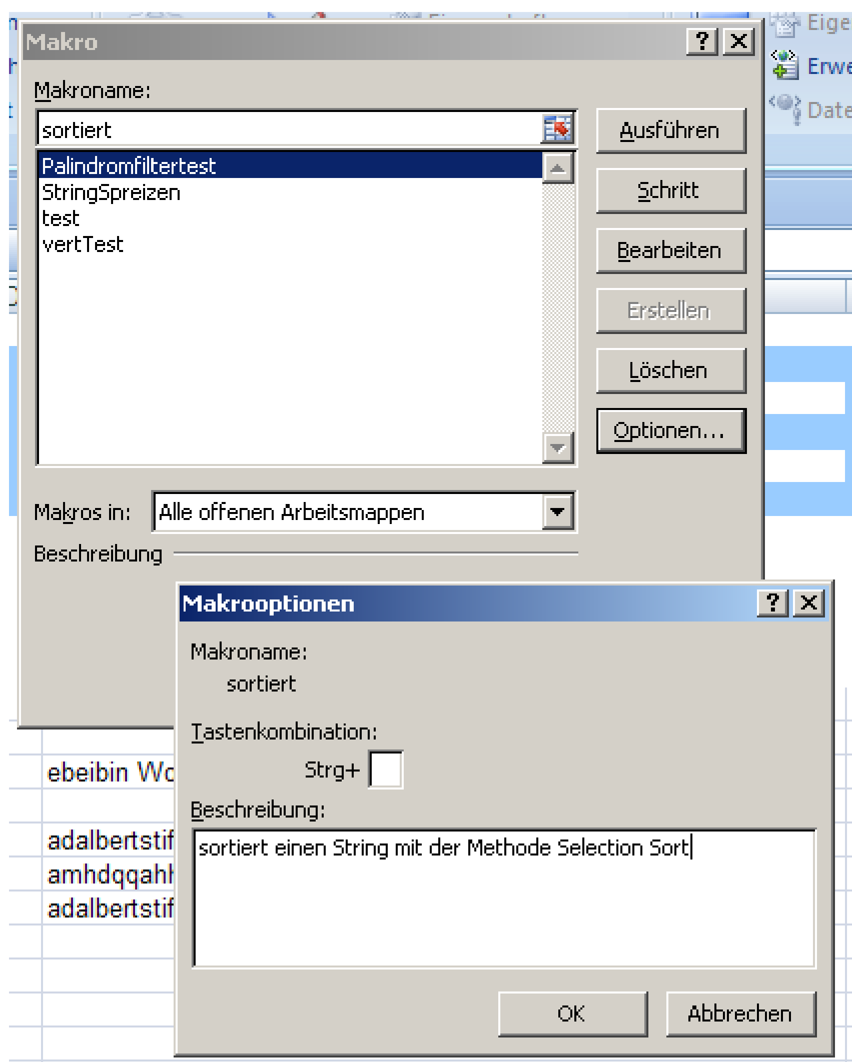
Die Flexibilität, die ParamArray bei der Parametereingabe erlaubt, ist beträchtlich. Der Input ist weder in Bezug auf die Anzahl der Einzelparameter noch bezüglich ihrer Art begrenzt. Da es sich um ein Variant-Array handelt, können die Einzelelemente des Arrays theoretisch jeden beliebigen Typ haben. Es wäre trotzdem verfehlt, ParamArray als „Allzweckwaffe“ zu verstehen. Was der Benutzer eingibt, muss auch korrekt verarbeitet werden!

2.11 Die Präsentation benutzerdefinierter Funktionen im Funktions- assistenten

Wie bereits erwähnt, wird eine fehlerlos programmierte Funktion, die in einem Standardmodul angesiedelt und als Public deklariert wurde, automatisch unter der Rubrik benutzerdefiniert im Funktionsassistenten angezeigt. In der Anzeige fehlt allerdings die kurze Erläuterung der Funktion, wie wir sie von den eingebauten Excel-Funktionen kennen.

Es ist recht einfach, einer UDF eine kurze Erläuterung hinzuzufügen. Dies geschieht in einem kurzen Dialog (s. Bild unten):

- Man wählt die Option Macros in der Registerkarte Entwicklertools. Es öffnet sich ein Fenster mit der Überschrift Makro. Nun gibt man den Namen der zu beschreibenden Funktion in das Feld mit der Überschrift Makroname ein (im Beispiel: sortiert).
- Man klickt auf die Schaltfläche Optionen. Es öffnet sich nun ein kleineres Fenster mit der Überschrift Makrooptionen.
- Nun gibt man die gewünschte Funktionsbeschreibung ein. Das Feld Tastenkombination lässt man frei (für UDF nicht anwendbar).
- Man verlässt das Fenster Makrooptionen mit <OK>.
- Das noch offene Fenster Makro verlässt man mit <Abbrechen<.



Wenn man eine UDF einer bestimmten Kategorie im Funktionsassistenten zuordnen will, ist das Vorgehen ein wenig komplizierter. Man muss hierfür eine kleine Prozedur schreiben und ausführen. Betrachten wir aber zunächst die verfügbaren Kategorien. Sie sind durchnummeriert:

1	Finanzmathematik
2	Datum & Zeit
3	Mathematik & Trigonometrie
4	Statistik
5	Matrix
6	Datenbank
7	Text
8	Logik
9	Information
10	Commands
11	Customizing
12	Macro Control
13	DDE/External
14	Benutzerdefinierte Funktionen
15 - 32	Eigene Kategorien

Nicht alle dieser Kategorien werden auch im Assistenten angezeigt. Gewöhnlich fehlen die Kategorien 10 bis 13 in der Anzeige, weil sie offensichtlich nicht benötigt werden.

Nun zu der Prozedur, mit der wir eine UDF einer Kategorie zuordnen und gleichzeitig eine Kurzerläuterung hinzufügen können. Die Prozedur macht Gebrauch von der Methode MacroOptions der Klasse Application des bereits erwähnten Excel-Objektmodells. Für jede Funktion, die wir platzieren wollen, fügen wir unserer Prozedur einen Aufruf der Methode MacroOptions hinzu. Mit der folgenden Prozedur UDFRegistrierung positionieren wir die beiden Funktionen sortiert und minPos, die bereits aus dem Abschnitt Zerlegung und Delegation bekannt sind. Die UDF sortiert wird einer neuen, selbst definierten Kategorie zugewiesen, die UDF minPos wird der bereits vorhandenen Kategorie mit der Nummer 7 (=Text) zugeordnet.

```
Public Sub UDFRegistrierung()

  Application.MacroOptions _
    Macro:="sortiert", _
    Description:="sortiert einen String entsprechend der ASCII-Tabelle", _
    Category:="eigene Funktionen"

  Application.MacroOptions _
    Macro:="minPos", _
    Description:="ermittelt das kleinste Zeichen in einem String" & _
             "entsprechend der ASCII-Tabelle", _
    Category:=7
End Sub
```

Die Prozedur kann in einem beliebigen Modul platziert und ausgeführt werden. Es genügt auch eine einzige Ausführung; es ist also nicht nötig, sie bei jedem Start von Excel von neuem auszuführen, wie in manchen Büchern behauptet wird. Sichtbar sind die Zuordnungen allerdings nur in der Arbeits-

mappe, in der die Prozedur ausgeführt wurde. Sollen die Zuordnungen in allen Arbeitsmappen wirksam werden, so muss man sie im Rahmen eines Add-Ins vornehmen (s. unten).

2.12 Benutzerdefinierte Funktionen als Add-In ausliefern

Ein Add-In ist eine Datei, die ein Benutzer seinem Excel-System hinzufügen kann, und die diesem System zusätzliche Funktionalität verleiht. Das Hinzufügen geschieht menügesteuert mit Hilfe des Add-In-Managers und ist schnell durchgeführt.

Add-Ins sind die ideale Form, mit Excel-VBA gestaltete Software auszuliefern. Die Entwickler programmieren wie gewohnt in einer ganz normalen Arbeitsmappe. Ist die Software fertig gestellt und ausgetestet, so wird sie mit Hilfe eines einfachen Verfahrens in ein Add-In verwandelt. Dieses Add-In kann dann an alle Bezieher der Software verteilt werden und wird von diesen mittels Add-In-Manager installiert. Es ist möglich (und üblich), den Quellcode der Software im Add-In gegenüber den Erwerbern zu verbergen und die Software auf diese Weise gegen Missbrauch zu schützen (s. nächster Abschnitt).

Eine Bibliothek von UDF als Add-In auszuliefern, ist besonders unproblematisch, weil ein solches Add-In im Excel-System der Empfänger keine einschneidenden Veränderungen hervorrufen wird, insbesondere keine Veränderungen in der Benutzeroberfläche. Die Installation des Add-In macht sich nur in einem zusätzlichen Angebot an UDF im Funktionsassistenten bemerkbar.

Das Vorgehen zum Erstellen eines Add-In wird im Folgenden nur grob beschrieben (für Excel 2007 und Excel 2010). Es ändert sich leicht von Version zu Version von Excel und ist im Übrigen im Internet und in der Literatur eingehend beschrieben (s. z.B. Walkenbach).

- Entwickle die Anwendung (hier: die UDF) und teste sie aus. Mache mindestens eine Sicherheitskopie im Format *.xlsm.
- Wechsle in die VBA-Entwicklungsumgebung und suche im Projektfenster (gewöhnlich links) das Projekt, von dem ein Add-In erzeugt werden soll. Klicke mit der rechten Maustaste darauf und wähle „Eigenschaften ...“. Es öffnet sich ein Fenster zur Eingabe der Projekteigenschaften.
- Trage in der Registerkarte Allgemein dieses Fensters in das Feld Projektname einen sinnvollen Namen ein. Formuliere dann eine kurze Beschreibung des Add-Ins im Feld Projektbeschreibung.
- Falls der Quellcode für den Erwerber verdeckt sein soll, so wähle die Registerkarte Schutz und gib dort ein Kennwort ein. Schließe dann das Fenster Projekteigenschaften mit <OK>.
- Gehe zu Excel und speichere die Arbeitsmappe im Add-In-Format (*.xlam). Der Speicherort ist beliebig.

Als Erwerber, der das Add-In erhält, können Sie es wie folgt installieren (Excel 2010):

- Klicke im Excel-Fenster auf Datei und wähle Optionen. Es öffnet sich ein Fenster mit der Überschrift Excel-Optionen.
- Wähle auf der linken Seite die Option Add-Ins. Die Anzeige im Fenster ändert sich.

- Auf der rechten Seite, ganz unten, wird eine ComboBox mit der Beschriftung „Verwalten“ angezeigt. Wähle „Excel-Add-Ins“ und Klicke auf <Gehe zu ...>.
- Es öffnet sich ein Fenster mit einer Liste von verfügbaren bzw. (markiert) bereits installierten Add-Ins. Falls das zu installierende Add-Ins bereits in der Liste erscheint, wähle es aus und verlasse das Fenster mit <OK>. Falls es nicht angezeigt wird, wähle <Durchsuchen> und suche es im Verzeichnis. Markiere es danach in der Liste und verlasse das Fenster mit <OK>.
- Die im Add-In enthaltenen UDF sollten nun im Funktionsassistenten sichtbar sein (in der Rubrik, dem sie vom Entwickler zugeordnet worden sind oder aber unter benutzerdefinierte Funktionen).

2.13 Den Code durch Passwort schützen

Wie der Quellcode durch ein Passwort geschützt werden kann, wurde bereits im vorhergehenden Abschnitt erläutert. Achten Sie darauf, das Passwort möglichst lang zu wählen und aus verschieden-artigen Zeichen (Klein- und Großbuchstaben, Zahlen, Sonderzeichen) zusammenzusetzen, damit es nicht geknackt werden kann.

2.14 Fallstudie magisches Quadrat

Ein magisches Quadrat ist eine n x n – Matrix, in der jede der ganzen Zahlen von 1 bis n^2 genau einmal vorkommt, und in der alle Spaltensummen, Zeilensummen und Diagonalensummen gleich sind.

Beispiel (n = 3):

8	1	6
3	5	7
4	9	2

Mit Hilfe des folgenden Vorgehens kann ein magisches Quadrat für jede beliebige ungerade und positive ganze Zahl n angelegt werden:

> Setze die 1 in die Mitte der ersten Zeile. Bei jeder der folgenden Zahlen gehe so vor:
>
> Die vorher gesetzte Zahl sei k. Um k + 1 zu setzen, gehe eine Zeile nach oben und eine Spalte nach rechts. Die so gefundene Position muss noch modifiziert werden, falls einer der folgenden drei Fälle vorliegt:
>
> - Falls die Bewegung über die oberste Zeile hinausführen würde, so setze k + 1 in die betreffende Spalte der untersten Zeile.
> - Falls die Bewegung über den rechten Rand hinausführen würde, so setze k + 1 in die erste Spalte der betreffende Zeile.
> - Falls die Bewegung zu einem bereits besetzten Feld hinführt oder über die rechte obere Ecke der Matrix hinaus, so setze k + 1 unmittelbar unter k.

Unsere Funktion magischesQuadrat nimmt die gewünschte Dimension der Matrix in Form eines Integer-Parameters n entgegen und liefert die daraus entwickelte Matrix als Integer-Array.

```
Public Function magischesQuadrat(ByVal n As Integer) As Integer()

  Dim i As Integer, j As Integer
  Dim z As Integer, s As Integer, nz As Integer, ns As Integer
  Dim q() As Integer
  ReDim q(1 To n, 1 To n)

  'Array mit Nullen vorbesetzen
  For i = 1 To n
    For j = 1 To n
        q(i, j) = 0
    Next j
  Next i

  'Array mit den endgültigen Werten besetzen
  z = 1
  s = n \ 2 + 1
  q(z, s) = 1
  For i = 2 To n * n
    nz = z - 1
    ns = s + 1
    If nz = 0 And ns = n + 1 Then     'Ausnahmen
        nz = z + 1
        ns = s
    ElseIf nz = 0 Then
        nz = n
    ElseIf ns = n + 1 Then
        ns = 1
    End If
    If q(nz, ns) <> 0 Then
        nz = z + 1
        ns = s
    End If
    q(nz, ns) = i                 'Wert i einfügen

    z = nz                        'für die nächste Runde
    s = ns
  Next i

  magischesQuadrat = q
End Function
```

Da es sich um eine Array-Funktion handelt, muss man beim Aufruf der Funktion aus einem Tabellenblatt heraus auch die Steuerungs- und die Umschalttaste drücken, wenn man die Eingabetaste betätigt. Außerdem ist darauf zu achten, dass genau n x n Zellen markiert sind.

2.15 Fallstudie Fehlerquadratsumme

Im Gegensatz zu der Funktion magischesQuadrat liefert die UDF ESS (für error sum-of-squares) dieser Fallstudie nur eine einzige Zahl. Sie ist unter mehreren Aspekten interessant:

- ESS verarbeitet einen Bereich aus einem Tabellenblatt, also ein Range.
- Es wird vom Prinzip der Zerlegung und Delegation Gebrauch gemacht, damit das Programm übersichtlich und wartbar bleibt. Das Gesamtprogramm besteht aus der Hauptfunktion ESS (für error sum-of-squares) und mehreren Hilfsfunktionen.
- Im Inneren der Funktion wird ein benutzerdefinierter Datentyp verwendet. Dieser Datentyp ist Public und in einem eigenen Modul platziert. Er wird auch für andere UDF verwendet, die in derselben Arbeitsmappe angesiedelt sind.
- Nur die Hauptfunktion ESS ist mit Public deklariert; die Hilfsfunktionen sind Private und deshalb im Funktionsassistenten nicht sichtbar.

Der Begriff der Fehlerquadratsumme

Die Fehlerquadratsumme ist eine Kennzahl, welche benutzt wird, um die Kompaktheit der Cluster einzuschätzen, welche bei einer Clusteranalyse gefunden werden. Stehen beispielsweise für eine Menge von Objekten zwei Clusteraufteilungen mit derselben Clusteranzahl zu Wahl, so kann man davon ausgehen, dass die mit der kleineren Fehlerquadratsumme die kompakteren Cluster aufweist.

Die Fehlerquadratsumme eines Clusters entspricht der Summe der Abstände (genauer: der quadrierten euklidischen Abstände) der in diesem Cluster befindlichen Objekte zum Zentrum des Clusters. Summiert man die Fehlerquadratsummen aller Cluster, so erhält man die Gesamtfehlerquadratsumme. Dies ist die Kennzahl, die unsere UDF ESS berechnet.

Für den quadrierten euklidischen Abstand $d(X_i, X_j)$ zwischen den Objekten X_i und X_j in einem p-dimensionalen Merkmalsraum gilt:

$$d(X_i, X_j) = \sum_{k=1}^{p} (x_{i_k} - x_{j_k})^2$$

Darin ist x_{ik} der Wert von X_i beim k-ten Merkmal.

Das Zentrum eines Clusters ist ein fiktiver Punkt im Merkmalsraum. Seine Merkmalswerte entsprechen den arithmetischen Mitteln aus den Merkmalswerten aller Objekte, die dem Cluster zugeordnet wurden.

Der Input der UDF

Die UDF ESS, welche die Gesamtfehlerquadratsumme ermittelt, hat die Kopfzeile

```
Public Function Fehlerquadratsumme (ByVal PopRng as Range) As Double
```

PopRng (für Population Range) ist ein Bereich einer Excel-Tabelle, welcher das Ergebnis der Clusteranalyse enthält. Die Zeilen der Tabelle entsprechen den geclusterten Objekten. Die letzte Spalte enthält die Clusterzuordnung, die davor liegenden Spalten enthalten die Merkmalswerte (Attributwerte) der Objekte. Das folgende Bild zeigt eine solche Tabelle für den Fall eines zweidimensionalen Merkmalsraums. Die Tabellenüberschrift dient dabei nur der Information der Benutzer. PopRng soll nur aus dem Rumpf der Tabelle bestehen (ohne Überschrift).

Die Anwendung von ESS ist auf numerische Merkmale beschränkt. Negative Werte und Kommazahlen sind erlaubt.

	A	B	C	D
1				
2		A1	A2	Cluster
3		2	2	C1
4		2	3	C1
5		2	6	C2
6		2	7	C2
7		3	2	C1
8		3	6	C2
9		3	7	C2
10		4	5	C3
11		5	5	C3
12		6	4	C3
13		6	5	C3
14		7	5	C3
15				

Der benutzerdefinierte Datentyp cluster

Die Arbeitsmappe, welche die UDF ESS enthält, enthält darüber hinaus noch weitere, hier nicht gezeigte Funktionen für andere Indizes, die zur Beurteilung der Qualität einer Clusteranalyse benutzt werden können. Alle diese Funktionen nehmen Bezug auf den Datentyp cluster. Dieser Datentyp ist deshalb in einem eigenen Modul angesiedelt und, da nicht als Private deklariert, in der ganzen Arbeitsmappe benutzbar.

In jeder der UDF der Arbeitsmappe wird zunächst der Funktionsinput, der Bereich PopRng, in ein Array des Typs cluster überführt, wobei jedes Element dieses Arrays einem Cluster entspricht. Diese Umwandlung erlaubt eine effizientere Verarbeitung des Inputs und erlaubt gleichzeitig, nebenbei „kostenlos“ einige Kenndaten des jeweiligen Clusters herauszuziehen. Im Einzelnen handelt es sich um den Namen des Clusters, die Anzahl der darin enthaltenen Objekte, die Koordinaten des Clusterzentrums und die Entfernung zwischen Clusterzentrum und dem am weitesten davon entfernten Objekt („Radius“). Die Komponente x ist ein zweidimensionales Array, welches die Merkmalswerte

der Clusterobjekte und, in der letzten Zeile, jene des Zentrums enthält. In der UDF ESS wird die Komponente maxDist nicht benötigt.

```
Type cluster
    x() As Double           'zweidimensional; für Objekte und Zentrum
    cname As String         'Clustername
    noOfObj As Long         'Anzahl der Objekte im Cluster
    maxDist As Double       'Distanz Mittelpunkt zu entferntestem Objekt
End Type
```

Die Hauptfunktion ESS

Die UDF ESS ist sehr kurz, weil die meisten Aufgaben an Unterfunktionen delegiert wurden. Man kann zwei Hauptschritte unterscheiden. Zunächst erfolgt die Überführung des Inputs PopRng in ein Array vom Typ cluster mit Hilfe der Funktion extractClusters, danach wird die Gesamtfehlerquadratsumme aus den Fehlerquadratsummen der einzelnen Cluster aufsummiert. Diese werden von der Funktion ClustESS geliefert.

```
Public Function ESS(ByVal PopRng As Range) As Double
  Dim i As Long
  Dim c() As cluster
  c = extractClusters(PopRng)
  ESS = 0
  For i = 1 To UBound(c)
    ESS = ESS + ClustESS(c(i).x)
  Next i
End Function
```

Beachten Sie, dass die Schleifenvariable I vom Typ Long ist. Damit können auch sehr große Inputbereiche verarbeitet werden.

Die Hilfsfunkionen

Den größten Aufwand bereitet die Funktion extractClusters, welche den Input vom Typ Range in eine Array des benutzerdefinierten Typs cluster überführt. Es lassen sich deutlich einige Etappen unterscheiden, die durch Kommentare hervorgehoben sind.

```
Private Function extractClusters(PopRng As Range) As cluster()

  Dim c() As cluster
  ReDim c(1 To 1)
  Dim i As Long, j As Long, k As Long
  Dim cid As Long              'ClusterId

  With PopRng

    'die Namen der Cluster erfassen und
    'Anzahl der Objekte je Cluster ermitteln
    c(1).cname = Trim(.Cells(1, .Columns.Count))
    c(1).noOfObj = 1
```

```
        For i = 2 To .Rows.Count
            cid = getCID(c, Trim(.Cells(i, .Columns.Count)))
            If cid < 1 Then
              ReDim Preserve c(1 To UBound(c) + 1)
              c(UBound(c)).cname = Trim(.Cells(i, .Columns.Count))
              c(UBound(c)).noOfObj = 1
            Else
              c(cid).noOfObj = c(cid).noOfObj + 1
            End If
        Next i

        'Wertematrizen der Cluster dimensionieren;
        'die letzte Zeile ist für den Mittelpunkt
        For i = 1 To UBound(c)
            ReDim c(i).x(1 To c(i).noOfObj + 1, 1 To .Columns.Count - 1)
        Next i

        'die Merkmalswerte übernehmen und Merkmalssummen
        'für Mittelwerte fortschreiben
        Dim objcount() As Long          'Zählerarray
        ReDim objcount(1 To UBound(c))  'für jedes Cluster einen Zähler
        For i = 1 To UBound(objcount)
            objcount(i) = 0
        Next i
        For i = 1 To .Rows.Count
          cid = getCID(c, .Cells(i, .Columns.Count))
          objcount(cid) = objcount(cid) + 1
          For j = 1 To .Columns.Count - 1
              c(cid).x(objcount(cid), j) = CDbl(.Cells(i, j))
              c(cid).x(c(cid).noOfObj + 1, j) = _
              c(cid).x(c(cid).noOfObj + 1, j) + c(cid).x(objcount(cid), j)
          Next j
        Next i

        'Mittelwerte: Attributsummen durch Objektzahl dividieren
        For i = 1 To UBound(c)              'alle Cluster
          For j = 1 To UBound(c(i).x, 2)    'alle Spalten
              c(i).x(UBound(c(i).x, 1), j) = _
              c(i).x(UBound(c(i).x, 1), j) / c(i).noOfObj
          Next j
        Next i

      End With

      extractClusters = c

    End Function
```

In der ersten Etappe von extractClusters wird PopRange Zeile für Zeile durchgegangen mit dem Ziel, die enthaltenen Cluster zu identifizieren und für jedes ein Element im Array c anzulegen. Hierbei hilft die Funktion getCID. Sie ermittelt aus dem aktuellen Zustand von c und einem Clusternamen, ob für diesen Namen bereits ein Element in c existiert. Falls dies der Fall ist, liefert getCID den Wert -1, falls

es sich aber um ein noch nicht berücksichtigtes Cluster handelt, wird eine ClusterId für ein neu in c anzulegendes Element geliefert.

```
Private Function getCID(ByRef c() As cluster, ByVal clname) As Long
  Dim found As Boolean
  Dim i As Long
  found = False
  i = 1
  Do While i <= UBound(c) And Not found
     If c(i).cname = clname Then found = True
     i = i + 1
  Loop
  getCID = IIf(found, i - 1, -1)
End Function
```

Wie bereits erwähnt, werden im zweiten Hauptschritt von ESS die Fehlerquadratsummen der einzelnen Cluster zur Gesamtfehlerquadratsumme aufsummiert. Die folgende Funktion ClustESS errechnet jeweils die Fehlerquadratsumme eines Clusters. Die Parametervariable c, welche aus einem zweidimensionalen Array besteht, repräsentiert dieses Cluster.

Im Wesentlichen besteht die Aufgabe daraus, die Distanzen zwischen den einzelnen Objekten des Clusters und dem Clusterzentrum aufzusummieren. Die hierfür notwendige Errechnung der quadrierten euklidischen Distanz zwischen einem Objekt und dem Clusterzentrum wird an eine weitere Hilfsfunktion distEuklQu delegiert (s. weiter unten).

```
Private Function ClustESS(ByRef c() As Double) As Double
  Dim i As Long, j As Long
  Dim o() As Double, cent() As Double
  Dim ESS As Double
  ESS = 0
  ReDim o(1 To UBound(c, 2))
  ReDim cent(1 To UBound(c, 2))
  For i = 1 To UBound(c, 2)
      cent(i) = c(UBound(c, 1), i)
  Next i
  For i = 1 To UBound(c, 1) - 1
    For j = 1 To UBound(c, 2)
        o(j) = c(i, j)
    Next j
    ESS = ESS + distEuklQu(o, cent)
  Next i
  ClustESS = ESS
End Function
```

```
Private Function distEuklQu(ByRef o1() As Double, ByRef o2() As Double) As Double
  Dim i As Integer
  If UBound(o1) <> UBound(o2) Then
    distEuklQu = -1
  Else
    distEuklQu = 0
    For i = 1 To UBound(o1)
        distEuklQu = distEuklQu + (o1(i) - o2(i)) ^ 2
```

```
      Next i
    End If
End Function
```

2.16 Fallstudie Matrizenmultiplikation

Die Matrizenmultiplikation steht zwar in Excel schon fix und fertig als eingebaute Funktion zur Verfügung, aber ihre Realisierung als UDF ist dennoch interessant, denn

- es handelt sich um eine Funktion, welche Bereiche verarbeitet und einen Bereich bzw. ein Array liefert,
- sie ist nicht trivial
- wir können an ihr demonstrieren, wie man unzulässige Parameterwerte abfangen kann
- man kann an ihr sehr gut die Einwickeltechnik demonstrieren.

Matrizen werden in der Programmierung als **zweidimensionales Array** dargestellt, dies ist in VBA auch nicht anders. Ein Problem mit den Arrays von VBA ist, dass man die Indexuntergrenzen frei wählen kann. Stellt man sich als Programmierer darauf ein, dass die zu verarbeitenden Arrays beliebige Indexuntergrenzen haben sollen, so werden die Matrixoperationen zum Teil sehr kompliziert.

Wir wollen daher vereinbaren, dass **alle Indizes bei 1 beginnen**. In Verbindung mit Matrizen ist dies auch dies auch in der Mathematik so üblich. Diese Vereinbarung soll sowohl für die Argumente gelten, die in die Matrixfunktionen eingehen, als auch für die Resultate dieser Funktionen.

Bei UDF, welche Matrizen verarbeiten oder liefern, haben wir das bereits mehrfach diskutierte Problem, dass die Verwendung des Datentyps Range einen universellen Einsatz der Funktionen verhindert. Wir wollen auch hier die oben gezeigte Einwickeltechnik verwenden und zunächst die Funktion in einer universell verwendbaren Form präsentieren. Erst danach wird sie so eingewickelt, dass sie auch als UDF taugt. Hier zunächst die universell verwendbare Version:

```
Public Function MProd(ByRef a() As Double, ByRef b() As Double) As Double()
  Dim p() As Double
  If UBound(a, 2) <> UBound(b, 1) Then      'Prüfung, ob Matrizen verkettet sind
    ReDim p(-1 To -1)
    MProd = p
    Exit Function
  End If
  ReDim p(1 To UBound(a, 1), 1 To UBound(b, 2))
  Dim i As Integer, j As Integer, k As Integer
  For i = 1 To UBound(a, 1)
    For j = 1 To UBound(b, 2)
      p(i, j) = 0
      For k = 1 To UBound(b, 1)
        p(i, j) = p(i, j) + a(i, k) * b(k, j)
      Next k
    Next j
  Next i
  MProd = p
End Function
```

Zunächst noch einige Erklärungen zu der If-Anweisung im oberen Teil:

Bei der Matrixmultiplikation ergibt sich ein Problem, das sich bei anderen Matrixoperationen in ähnlicher Form stellt. Die Operation ist an bestimmte Voraussetzungen geknüpft. Bei der Multiplikation A x B ist es die Bedingung, dass A und B **verkettet** sein müssen, dass also die Spaltenzahl von A gleich der Zeilenzahl von B sein muss.

Wie kann man nun bei Verletzung der Bedingungen der aufrufenden Stelle mitteilen, dass ein **Verstoß gegen die Voraussetzungen** vorliegt? Eine gängige Methode ist, dass die Funktion in einem solchen Fall Werte liefert, die normalerweise ausgeschlossen sind. In der folgenden Lösung wird der Fehler über die Indexgrenzen des gelieferten Arrays signalisiert: sind die beiden Matrizen nicht verkettbar, so wird ein Array geliefert, bei dem sowohl die Indexuntergrenze als auch die Indexobergrenze -1 ist.

Für das **Einwickeln** benötigen wir die Hilfsfunktion RangeToDblArray, welche die Parameter a und b in Arrays vom Typ Double verwandeln kann:

```
Public Function RangeToDblArray(ByVal r As Range) As Double()
  Dim d() As Double
  ReDim d(1 To r.Cells.Rows.Count, 1 To r.Cells.Columns.Count)
  Dim i As Integer, j As Integer
  For i = 1 To UBound(d, 1)
    For j = 1 To UBound(d, 2)
        d(i, j) = CDbl(r.Cells(i, j).Value)
    Next j
  Next i
  RangeToDblArray = d
End Function
```

Mit ihrer Hilfe formulieren wir die einwickelnde Funktion (wrapper) folgendermaßen:

```
Public Function MProdCW(ByVal a As Range, ByVal b As Range) As Double()
  MProdCW = MProd(RangeToDblArray(a), RangeToDblArray(b))
End Function
```

Wer eine alte Version von Excel verwendet, die vielleicht den Ergebnistyp Double() nicht akzeptiert, kann ersatzweise auch den Ergebnistyp Variant verwenden:

```
Public Function MProdCW(ByVal a As Range, ByVal b As Range) As Variant
  MProdCW = MProd(RangeToDblArray(a), RangeToDblArray(b))
End Function
```

Eine explizite Typenumwandlung in den Ergebnistyp Variant wäre zwar möglich, denn es gibt hierfür eine Umwandlungsfunktion CVar, aber dies ist nicht nötig, denn Variant ist ja eigentlich kein Datentyp.

2.17 Fallstudie Spaltensummen: Text und Zahlen im Funktionsergebnis

Die Funktion, die wir in diesem Abschnitt betrachten, ist eher kurz und übersichtlich, besitzt aber eine Eigenart, die sie von den bisher betrachteten Funktionen unterscheidet: Das Funktionsergebnis besteht aus einem Array, das sowohl Zahlen als auch Text enthält. Außerdem wird die bereits bekannte (Abschnitt 2.7) behandelte Technik des Einwickelns benutzt, um aus einer nützlichen Funktion ohne großen Aufwand eine zweite herzustellen.

Zweck der Funktion ist, die Summen aller Spalten einer Wertetabelle auf einmal zu liefern. Normalerweise würden wir hierzu die eingebaute Excel-Funktion Summe benützen, aber diese müssten wir für jede Spalte einzeln anwenden. Unsere Funktion soll dagegen die Summen nebeneinander in einem Array liefern.

Die erste Fassung der Funktion liefert ein eindimensionales Array mit allen Spaltensummen. Die Spalten in diesem Summenarray sind nicht beschriftet. Man könnte die Funktion so benutzen, dass man das Array direkt unter die Wertetabelle setzt, wie im folgenden Bild gezeigt:

x1	x2	x3
10,0	50,5	36,0
77,6	25,0	80,0
50,0	65,0	92,0
58,0	59,0	90,0
79,0	26,0	79,1
10,0	55,0	30,0
15,0	55,8	31,7
17,0	61,0	40,0
60,0	63,0	94,0
63,0	67,0	92,0
77,0	27,0	81,0
80,5	21,4	83,0
75,0	22,0	82,0
65,0	64,0	96,0
70,0	68,0	97,0
807,1	729,7	1103,8

Eine Beschriftung ist hier auch gar nicht nötig, weil die Beschriftungen der über den Summen liegenden Wertetabelle auch für die Summen gelten.

Die Funktion Spaltensummen, welche die Summen liefert, nimmt die Wertetabelle als Parameter entgegen (ohne Überschriften) und liefert ein Double-Array. Zur Berechnung der Summen der einzelnen Spalten wird auf die Excel-Funktion Summe (WorksheetFunction Sum) zurückgegriffen:

```
Public Function Spaltensummen(ByVal rng As Excel.Range) As Double()
  Dim i As Integer
  Dim n As Integer                    'Anzahl der Attribute
  n = rng.Columns.Count
  Dim s() As Double
  ReDim s(1 To n)
  For i = 1 To n
```

```
        s(i) = Application.WorksheetFunction.Sum(rng.Columns(i))
    Next i
    Spaltensummen = s
End Function
```

Falls wir die Spaltensummen nicht direkt unter die Wertetabelle setzen wollen, sondern an irgendeine andere Stelle der Arbeitsmappe, benötigen wir Überschriften. Hierzu wollen wir eine Funktion schreiben, welche die Summen mit Spaltenüberschriften liefert.

x1	x2	x3
807,1	729,7	1103,8

Um nicht alles von Grund auf neu programmieren zu müssen, verwenden wir die Einwickeltechnik und nehmen die oben gezeigte Funktion Spaltensummen als Basis. Die neue, umfassende Funktion heißt SpaltensummenM und benötigt als Eingabeparameter den Bereich der Wertetabelle einschließlich der Überschrift:

```
Public Function SpaltensummenM(ByVal rngM As Excel.Range) As Variant
    Dim i As Integer
    Dim a() As Variant              'für Beschriftungen und Summenwerte
    Dim s() As Double               'nur für die Summen
    ReDim a(1 To 2, 1 To rngM.Columns.Count)
    'Die Spaltensummen werden abgerufen und in s abgelegt
    Dim rngV As Excel.Range
    Set rngV = rngM.Range(Cells(2, 1), Cells(rngM.Rows.Count, rngM.Columns.Count))
    s = Spaltensummen(rngV)
    'die Werte werden nach a übertragen
    For i = 1 To rngM.Columns.Count
        a(1, i) = rngM.Cells(1, i).Value
        a(2, i) = s(i)
    Next i
    SpaltensummenM = a
End Function
```

Intern benutzt die Funktion zwei Arrays, um die zu liefernden Werte zusammen zu stellen. Das Double-Array s nimmt die Spaltensummen auf, das Array a vom Typ Variant dient der Aufnahme aller zu liefernden Daten, also der Summen und der Spaltenbeschriftungen.

Um die Spaltensummen zu gewinnen, wird zunächst der Teil des Eingabebereichs, der die Zahlen enthält, von der Überschrift abgespalten und in der Variablen rngV gespeichert. Dann können die Spaltensummen durch Aufruf der Funktion Spaltensummen abgerufen und in s gespeichert werden, wobei rngV der Funktion als Parameter übergeben wird.

In der folgenden Schleife werden sowohl die Überschriften als auch die in s befindlichen Summen in das Array übertragen.

Beachten Sie, dass die im Funktionsinneren verwendete Variable a ein Array vom Typ Variant ist, wogegen das Ergebnis der Funktion zwar ebenfalls als Variant, aber nicht als Array deklariert ist. Wenn Sie dies unlogisch finden sollten, sind Sie nicht der einzige.

Mögliche Erweiterungen

Richtig nützlich wird die Funktion, wenn sie neben der Summen noch andere Spaltenkennzahlen liefert, wie z.B. den Durchschnitt, das Minimum, das Maximum und die Standardabweichung. In diesem Fall sollte man die zu liefernde Matrix um eine Beschriftungsspalte für die Zeilen ergänzen.