Tricks of the Trade

This is a column of programming tricks and techniques. You are encouraged to submit ideas to this column.

Edited by Paul Abbott

Searching for Options

Suppose you would like to find out which functions have a WorkingPrecision option. Direct information on WorkingPrecision is not all that helpful:

```
In[1]:=?WorkingPrecision

WorkingPrecision is an option for various numerical operations which specifies how many digits of precision should be maintained in internal computations.

WorkingPrecision -> n causes all internal computations to be done to at most n-digit precision.

Attributes[WorkingPrecision] = {Protected}
```

One approach is to scan through the options of all objects in the System` context using the Names command to find a complete list of names:

```
In[2]:=?Names

Names["string"] gives a list of the names of symbols which match the string. Names["string", SpellingCorrection->True] includes names which match after spelling correction.

In[3]:= Names["System`*"] // Short

Out[3]//Short=
{Abort, $Aborted, AbortProtect, <<1128>>, ZeroTest, Zeta}
```

It is easily determined that there are

```
In[4]:= Length[allnames]

Out[4]= 1131
```

names in the System` context. Each element in the list returned by Names is a string, for example:

```
In[5]:= First[allnames] // FullForm

Out[5]//FullForm= "Abort"
```

However, the Options function expects a Symbol or a stream as its first argument:

```
In[6]:= ?Options

Options[symbol] gives the list of default options assigned to a symbol. Options[expr] gives the options explicitly specified in a particular expression such as a graphics object. Options[stream] or Options["sname"] gives options associated with a particular stream. Options[expr, name] gives the setting for the option name in an expression. Options[expr, {name1, name2,...}] gives a list of the settings for the options namei.
```

The function ToExpression converts a String to a Symbol, enabling the determination of the Options for all names in the system context:

```
In[7]:= (alloptions = Options /@ ToExpression /@ allnames) // Short[#,

Out[7]//Short= {{}, {}, {}, {}, {}, {}, {}, {}, {},
{DigitBlock -> Infinity, NumberPoint -> ., SignPadding -> False, <<4>>, NumberSeparator -> ,, ExponentFunction -> Automatic}, {}, <<1101>>, {}, {},
{IncludeSingularTerm -> False}}
```

It is now easy to find the locations of all options involving WorkingPrecision:

```
In[8]:= Position[alloptions, WorkingPrecision]

Out[8]= {{19, 7, 1}, {250, 7, 1}, {322, 6, 1}, {333, 6, 1},
{574, 7, 1}, {638, 7, 1}, {645, 9, 1}, {655, 8, 1},
{663, 8, 1}, {823, 7, 1}, {929, 7, 1}, {930, 6, 1}}
```

From this result one can determine a list of functions with a WorkingPrecision option:

```
In[9]:= ToExpression /@ allnames[[First /@ %]]

Out[9]= {AlgebraicRules, Eliminate, FindMinimum, FindRoot,
MainSolve, NDSolve, NIntegrate, NProduct, NSum, Reduce, Solve, SolveAlways}
```
and then pair up each function with its WorkingPrecision option:

```
In[10]:= {#, Options[#, WorkingPrecision]}& /@ %
```

```
Out[10]= {{AlgebraicRules, {WorkingPrecision -> Infinity}},
         {Eliminate, {WorkingPrecision -> Infinity}},
         {FindMinimum, {WorkingPrecision -> 19}},
         {FindRoot, {WorkingPrecision -> 19}},
         {MainSolve, {WorkingPrecision -> Infinity}},
         {NDSolve, {WorkingPrecision -> 19}},
         {NIntegrate, {WorkingPrecision -> Infinity}},
         {NProduct, {WorkingPrecision -> 19}},
         {NSum, {WorkingPrecision -> 19}},
         {Reduce, {WorkingPrecision -> Infinity}},
         {Solve, {WorkingPrecision -> Infinity}},
         {SolveAlways, {WorkingPrecision -> Infinity}}}
DSolve
Mathematica does not directly solve the following linear ordinary differential equation:

In[1]:= eqns = {y''[r] + (1/r) y'[r] == c, y'[a] == 0, y'[b] == e + d y[b]}
Out[1]= {y''[r]/r + y'[r] == c, y'[a] == 0, y'[b] == e + d y[b]}

In[2]:= DSolve[eqns, y[r], r]
Out[2]= DSolve[{y''[r]/r + y'[r] == c, y'[a] == 0, y'[b] == e + d y[b]}, y[r], r]

However, the solution is immediate after loading the Version 2.2 package DSolve:

In[3]:= << Calculus`DSolve`
In[4]:= DSolve[eqns, y[r], r]
Out[4]=

\[
\begin{align*}
\frac{2}{a c (1 - b d \log[b])} \left( \frac{c r}{4} \right) - \frac{2}{b d} \left( \frac{a c \log[r]}{2} \right)
\end{align*}
\]

As a check, using the assignment:

In[5]:= y[r_] = y[r] /. First[%]
Out[5]=

\[
\begin{align*}
\left( \frac{2}{a c (1 - b d \log[b])} \right) \left( \frac{c r}{4} \right) - \frac{2}{b d} \left( \frac{a c \log[r]}{2} \right)
\end{align*}
\]

one sees that all the equations are satisfied:

In[6]:= eqns // Simplify
Out[6]= {True, True, True}

It is a good idea to clear \textit{y} after this calculation has been completed:

In[7]:= Clear[y]

CollectCases
Suppose we want to simplify the following expression:


One way to collect all terms involving \textit{Cos} is to use \textit{Collect} and specify the terms explicitly:

In[2]= \text{Collect[exp, }\text{Cos[...]}\text{]}
Out[2]= (a + b) Cos[x] + (c + d) Cos[2 x] + (e + f) Cos[x - y]

Unfortunately, \textit{Collect} does not work with patterns:

In[3]= \text{Collect[exp, }\text{Cos[_]}\text{]}

The function \textit{Cases} works with patterns, and it can be used to identify all terms involving \textit{Cos}:

In[4]= \text{Cases[exp, }\text{Cos[...]}\text{, Infinity}]/\text{Union}
Out[4]= \{Cos[x], Cos[2 x], Cos[x - y]\}

or just those that have are the \textit{Cos} of a sum of terms:

In[5]= \text{Cases[exp, }\text{Cos[Plus]}\text{, Infinity}]/\text{Union}
Out[5]= \{Cos[x - y]\}

Constructing an operator that collects terms using patterns is now straightforward:

In[6]= \text{CollectCases[exp, }h_\text{]} := \text{Collect[exp, Cases[exp, }h_\text{, Infinity}]/\text{Union]}

Now we can easily collect all terms involving \textit{Cos}:

In[7]= \text{CollectCases[exp, }\text{Cos[...]}\text{]}
Out[7]= (a + b) Cos[x] + (c + d) Cos[2 x] + (e + f) Cos[x - y]

Here are examples of two other patterns:

In[8]= \text{CollectCases[exp, }\text{Cos[2]}\text{]}

In[9]= \text{CollectCases[exp, }\text{Cos[a] + b]}\text{]}

Implicit Differentiation
Consider the problem of finding the slope of a curve at a particular point. For example, the curve

\[ \text{curve}[x_, y_] = x^3 - 3 x y^2 + y^3 == 1; \]

passes through the point \((2, -1)\):

In[1]= \text{curve[2, -1]}

Implicit differentiation can be achieved by taking the (total) derivative of the equation:

In[3]= \text{Dt[curve[x, y], x]}
Out[3]= 2 x^2 - 3 y^2 - 6 x y y'[y, x] + 3 y^2 y'[y, x] == 0

40 THE MATHEMATICA JOURNAL © 1994 Miller Freeman Publications
After solving for $Dt[y, x]$:

```
In[4]: = Solve[X, Dt[y, x]]
Out[4]= {(Dt[y, x] -> -(\(-x\) \(-iy\))\(-2\ \(x\) \(+\(y\)))\(-2\ \(x\) \(+\(y\)))}
```

one finds the general formula for the slope:

```
In[5]: = slope[x_, y_] = Dt[y, x] /. First[X]
Out[5]= -(\(-x\) \(-iy\))\(-2\ \(x\) \(+\(y\)))\(-2\ \(x\) \(+\(y\)))
```

It is now easy to compute the value of the derivative at the given point:

```
In[6]: = slope[2, -1]
Out[6]= -\(-3\)
```

### Using Padé to Generate Code

Suppose you require efficient Fortran or C code to compute a large number of values of a particular special function. One method, suggested by Bardo Muller (bardo@ief-paris-sud.fr), is to use Padé approximants to give an accurate rational approximation for the function.

Consider Dawson's integral defined by

```
In[1]: = dawson[x_] = Exp[-x^2] Integrate[Exp[y^2], {y, 0, x}]
```

```
Out[1]= \(-\(\sqrt{\pi}\) \(\text{Erfi}[x]\)\(-2\ \(x\) \(+\(2\))\)
```

Most Fortran or C compilers do not include code for the evaluation of the imaginary error function:

```
In[2]: = ?Erfi
```

Erfi[z] gives the imaginary error function

```
\(\text{erfi}(z) == -i \text{erf}(i z)\).
```

Nevertheless, after loading

```
In[3]: = << Calculus`Pade`
```

```
In[4]: = dawson34[x_] = Padé[dawson[x], {x, 0, 3, 4}]
Out[5]= \(-\(2\ \(x\) \(+\(2\))\)
```

```
\(-\(4\ \(x\) \(+\(2\))\)
```

```
\(-\(4\ \(x\) \(+\(2\))\)
```

```
\(-\(3\ \(x\) \(+\(1\))\)
```

```
\(-\(1\ \(x\) \(+\(1\))\)
```

```
\(-\(1\ \(x\) \(+\(1\))\)
```

```
\(-\(1\ \(x\) \(+\(1\))\)
```

```
\(-\(1\ \(x\) \(+\(1\))\)
```

```
\(-\(1\ \(x\) \(+\(1\))\)
```

```
\(-\(1\ \(x\) \(+\(1\))\)
```

```
\(-\(1\ \(x\) \(+\(1\))\)
```

```
\(-\(1\ \(x\) \(+\(1\))\)
```

Comparing dawson34 with the exact integral, one finds that this approximation is accurate to better than 0.15% over the interval [-1, 1]:

```
In[6]: = Plot[dawson[x] - dawson34[x], {x, -1, 1}, PlotRange -> All];
```

Thus, dawson34 should be appropriate for single precision on many platforms.

In many cases, one can use Horner’s rule to improve efficiency and avoid underflow (see Tricks of the Trade, Volume 2, issue 2):

```
In[7]: = Horner[p_?PolynomialQ, x_]:=
    Fold[x #1 + #2 &, 0, Reverse[CoefficientList[p, x]]]
```

Applying Horner to dawson34, one obtains:

```
In[8]: = Horner[Numerator[#], x] / Horner[Denominator[#], x] & @ dawson34[x]
```

```
Out[10]= \(-\(2\ \(x\) \(+\(1\))\)
```

```
\(-\(2\ \(x\) \(+\(1\))\)
```

```
\(-\(2\ \(x\) \(+\(1\))\)
```

```
\(-\(2\ \(x\) \(+\(1\))\)
```

```
\(-\(2\ \(x\) \(+\(1\))\)
```

```
\(-\(2\ \(x\) \(+\(1\))\)
```

```
\(-\(2\ \(x\) \(+\(1\))\)
```

```
\(-\(2\ \(x\) \(+\(1\))\)
```

```
\(-\(2\ \(x\) \(+\(1\))\)
```

```
\(-\(2\ \(x\) \(+\(1\))\)
```

```
\(-\(2\ \(x\) \(+\(1\))\)
```

VOLUME 4, ISSUE 1 41

```
Finally, one can use \texttt{FortranForm} to produce a code fragment suitable for computing Dawson's integral numerically over the range \([-1, 1] \): 

\begin{verbatim}
In[1]:= FortranForm[N[N[]]]
Out[1]/FortranForm=
  x*(1. + x**2*(-0.1333333333333333333 +
    - x**2*(0.03418803418803418803 +
    - 0.00101212121212121*x**2)))/
  (1. + x**2*(0.5333333333333333333 +
    - x**2*(0.1230769230769230769 +
    - 0.001101121101121101121*x**2)))
\end{verbatim}

(See also the article by Mark Sofroniou in Volume 3, issue 3.)

\section*{Simplifying a Sum}

Consider the summation

\[ f(x) = \sum_{k=-\infty}^{\infty} \frac{1}{(k+x)^2 + 1} \]

where \( x \) is real. It is apparent that \( f(x) \) is real, positive, and that it is periodic in \( x \) with period unity. After loading

\begin{verbatim}
In[1]:= \<< Algebra\`SymbolicSum\`
\end{verbatim}

it is straightforward to find a closed form for \( f \):

\begin{verbatim}
In[2]:= Sum[1/((x+k)^2 + 1), {k, -Infinity, Infinity}]
Out[2]= I Pi Cot[I (-1 + I + x)] - I Pi Cot[I (1 - I - x)]
\end{verbatim}

However, this expression contains complex variables, even though \( f(x) \) is explicitly real. Using \texttt{ComplexExpand} does not make things much clearer:

\begin{verbatim}
In[3]:= ComplexExpand[%]
Out[3]= \(-Pi Sin[Pi (-1 + I + x)]\)
  \\(-Pi Sin[Pi (1 - I - x)]\)
  \\(-2 (Cos[2 Pi (-1 + x)] - Cosh[2 Pi])\)
  \\(-2 (Cos[2 Pi (1 - x)] - Cosh[2 Pi])\)
\end{verbatim}

But expanding the trig functions using

\begin{verbatim}
In[4]:= Expand[%, Trig -> True] // ExpandAll
  \\(-Cos[2 Pi x] - Cosh[2 Pi]\)
\end{verbatim}

yields an explicitly real closed form for \( f(x) \). The expected periodicity is evident through the \( \cos[2 \pi x] \) term.

\section*{Testing Pattern Matching}

A useful trick to test pattern-matching is to use a replacement rule with a condition (such as \texttt{Print}) that can never evaluate to True. For example:

\begin{verbatim}
In[1]:= \{a, b, c\} /. \{x_, y_\} :> Anything /;
Print["Trying ", x -> \{\}, y -> \{\}\]
Out[1]= \{a, b, c\}
\end{verbatim}

Here, the pattern does not match the expression. However, if

\begin{verbatim}
In[2]:= Alias[\"\"]
Out[2]= Blank
\end{verbatim}

is replaced by

\begin{verbatim}
In[3]:= Alias[\"\""]
Out[3]= BlankSequence
\end{verbatim}

in one of the pattern variables, then there is one possible match:

\begin{verbatim}
In[4]:= \{a, b, c\} /. \{x___, y___\} :> Anything /;
Print["Trying ", x -> \{\}, y -> \{\}\]
\end{verbatim}

Trying \( x \) \( \rightarrow \) \{\}, \( y \) \( \rightarrow \) \{\}

\begin{verbatim}
Out[4]= \{a, b, c\}
\end{verbatim}

The general pattern involving

\begin{verbatim}
In[5]:= Alias[\"\""]
\end{verbatim}

takes into account all possible matches:

\begin{verbatim}
In[6]:= \{a, b, c\} /. \{x___, y___\} :> Anything /;
Print["Trying ", x \( \rightarrow \) \{\}, y \( \rightarrow \) \{\}\]
\end{verbatim}

Trying \( x \) \( \rightarrow \) \{}, \( y \) \( \rightarrow \) \{\}

\begin{verbatim}
Out[6]= \{a, b, c\}
\end{verbatim}