

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]:= (allnames = 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[#, 5]&
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[aloptions, WorkingPrecision]
Out[8]= {{19, 7, 1}, {250, 7, 1}, {332, 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 -> 19}},
  {NProduct, {WorkingPrecision -> 19}},
  {NSum, {WorkingPrecision -> 19}},
  {Reduce, {WorkingPrecision -> Infinity}},
  {Solve, {WorkingPrecision -> Infinity}},
  {SolveAlways, {WorkingPrecision -> Infinity}}}
```

Combining these steps into a function:

```
In[11]:= FunctionsWithOption[opt_] :=
  {#, Options[#, opt]}& /@ ToExpression /@
  allNames[First /@ Position[allOptions, opt]]]
```

here are all functions that have a `PlotPoints` option:

```
In[12]:= FunctionsWithOption[PlotPoints]
Out[12]= {{ContourPlot, {PlotPoints -> 15}},
  {DensityPlot, {PlotPoints -> 15}},
  {ParametricPlot, {PlotPoints -> 25}},
  {ParametricPlot3D, {PlotPoints -> Automatic}},
  {Plot, {PlotPoints -> 25}},
  {Plot3D, {PlotPoints -> 15}}}
```

New Packages in Version 2.2

Elliptic Integration

Mathematica does not compute the following integrals directly:

$$\int \frac{dx}{\sqrt{1+x^3}}, \quad \int \sqrt{\tan x} dx,$$

The first integral may be recognizable as an *elliptic integral*. After a change of variables:

```
In[1]:= Sqrt[Tan[x]] Dt[x] /. x -> ArcTan[u^2] // PowerExpand
Out[1]=  $\frac{2 u^2 Dt[u]}{1+u^4}$ 
```

the second integral may be identified as a pseudo-elliptic integral. Both cases are handled by the package `EllipticIntegrate`, distributed with Version 2.2:

```
In[2]:= << Calculus`EllipticIntegrate`
In[3]:= Integrate[Sqrt[Tan[x]], x]
Out[3]= (Csc[x] (-ArcSin[Cos[x] - Sin[x]] -
  Log[Cos[x] + Sin[x] + Sqrt[Sin[2 x]]])
  Sqrt[Sin[2 x]] Sqrt[Tan[x]]) / 2
```

```
In[4]:= Integrate[1/Sqrt[1 + x^3], x]
Out[4]= -((Sqrt[3] (1 + x)
  (1 + Sqrt[3] + x)^2
  Sqrt[ $\frac{1-x+x^2}{(1+\sqrt{3}+x)^2}$ ]
  EllipticF[ArcCos[ $\frac{1-\sqrt{3}+x}{1+\sqrt{3}+x}$ ],  $\frac{2+\sqrt{3}}{4}$ ]) /
  (Sqrt[3] Sqrt[1 + x^3]))
```

Note that the function `EllipticF` appears in the result:

```
In[5]:= ?EllipticF
EllipticF[phi, m] gives the elliptic integral of the first
kind F(phi|m).
```

Taking the derivative:

```
In[6]:= D[EllipticF[phi, m], phi]
Out[6]=  $\frac{1}{\text{Sqrt}[1 - m \text{Sin}[\text{phi}]^2]}$ 
```

reveals that:

$$F(\phi|m) = \int \frac{d\phi}{\sqrt{1 - m \sin^2 \phi}}$$

(using the notation of Chapter 17.2 of *Handbook of Mathematical Functions*, edited by M. Abramowitz and I. Stegun).

Some integrals are still not handled directly:

```
In[7]:= Integrate[Tan[x]^(1/3), x]
Out[7]= Integrate[Tan[x]^(1/3), x]
```

but a change of variables helps:

```
In[8]:= Tan[x]^(1/3) Dt[x] /. x -> ArcTan[u]
Out[8]=  $\frac{u^{1/3} Dt[u]}{1+u^2}$ 
In[9]:= Integrate[%/Dt[u], u]
Out[9]=  $\frac{-(\text{Sqrt}[3] \text{ArcTan}[\text{Sqrt}[3] - 2 u^{1/3}])}{2} -$ 
 $\frac{\text{Sqrt}[3] \text{ArcTan}[\text{Sqrt}[3] + 2 u^{1/3}] \text{Log}[1 + u^{2/3}]}{2} +$ 
 $\frac{\text{Log}[1 - \text{Sqrt}[3] u^{1/3} + u^{2/3}]}{4} +$ 
 $\frac{\text{Log}[1 + \text{Sqrt}[3] u^{1/3} + u^{2/3}]}{4}$ 
```

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] == d y[b] + e}
Out[1]:= {y'[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]
                + 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]:= {{y[r] -> (2 b c - b^2 c d - 4 e) / (4 d) + (c r^2) / (4 d) -
             (a^2 c (1 - b d Log[b]) - a^2 c Log[r]) / (2 b d)}
```

As a check, using the assignment:

```
In[5]:= y[r_] = y[r] /. First[%]
Out[5]:= (2 b c - b^2 c d - 4 e) / (4 d) + (c r^2) / (4 d) -
          (a^2 c (1 - b d Log[b]) - a^2 c Log[r]) / (2 b d)
```

one sees that all the equations are satisfied:

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

It is a good idea to clear `y` after this calculation has been completed:

```
In[7]:= Clear[y]
```

CollectCases

Suppose we want to simplify the following expression:

```
In[1]:= exp = a Cos[x] + b Cos[x] + c Cos[2 x] +
           d Cos[2 x] + e Cos[x - y] + f Cos[x - y]
Out[1]:= a Cos[x] + b Cos[x] + c Cos[2 x] + d Cos[2 x] +
           e Cos[x - y] + f Cos[x - y]
```

One way to collect all terms involving `Cos` is to use `Collect` and specify the terms explicitly:

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

Unfortunately, `Collect` does not work with patterns:

```
In[3]:= Collect[exp, Cos[_]]
Out[3]:= a Cos[x] + b Cos[x] + c Cos[2 x] + d Cos[2 x] +
           e Cos[x - y] + f Cos[x - y]
```

The function `Cases` works with patterns, and it can be used to identify *all* terms involving `Cos`:

```
In[4]:= Cases[exp, Cos[_], Infinity] // Union
Out[4]:= {Cos[x], Cos[2 x], Cos[x - y]}
```

or just those that have are the `Cos` of a sum of terms:

```
In[5]:= Cases[exp, Cos[_Plus], Infinity] // Union
Out[5]:= {Cos[x - y]}
```

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

```
In[6]:= CollectCases[exp_, h_] :=
        Collect[exp, Cases[exp, h, Infinity] // Union]
```

Now we can easily collect all terms involving `Cos`:

```
In[7]:= CollectCases[exp, Cos[_]]
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]:= CollectCases[exp, Cos[2_]]
Out[8]:= a Cos[x] + b Cos[x] + (c + d) Cos[2 x] + e Cos[x - y] +
           f Cos[x - y]
```

```
In[9]:= CollectCases[exp, Cos[a_ + b_]]
Out[9]:= a Cos[x] + b Cos[x] + c Cos[2 x] + d Cos[2 x] +
           (e + f) Cos[x - y]
```

Implicit Differentiation

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

```
In[1]:= curve[x_, y_] = x^3 - 3x y^2 + y^3 == 1;
```

passes through the point (2, -1):

```
In[2]:= curve[2, -1]
Out[2]:= True
```

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

```
In[3]:= Dt[curve[x, y], x]
Out[3]:= 3 x^2 - 3 y^2 - 6 x y Dt[y, x] + 3 y^2 Dt[y, x] == 0
```

After solving for Dt[y, x]:

```
In[4]:= Solve[%, Dt[y,x]]
```

```
Out[4]= {{Dt[y, x] -> -((x^2 - y^2) / (2 x y + y^2))}}
```

one finds the general formula for the slope:

```
In[5]:= slope[x_, y_] = Dt[y, x] /. First[%]
```

```
Out[5]= -(x^2 - y^2) / (2 x y + y^2)
```

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

```
In[6]:= slope[2, -1]
```

```
Out[6]= -(3/5)
```

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] Erfi[x]) / (2 E^x)
```

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
erfi(z) == -i erf(i z).
```

Nevertheless, after loading

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

it is straightforward to use Pade to produce a rational function that approximates Dawson's integral quite well near $x = 0$:

```
In[4]:= ?Pade
```

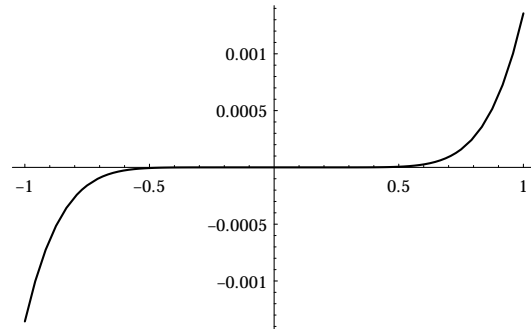
```
Pade[func, {x, x0, m, k}] gives the Pade' approximation to
func (a function of the variable x) where the constant
x0 is the center of expansion and m and k are the
degrees of the numerator and denominator, respectively.
```

```
In[5]:= dawson34[x_] = Pade[dawson[x], {x, 0, 3, 4}]
```

```
Out[5]= (x - (2 x^3) / 21) / (1 + (4 x^2) / 7 + (4 x^4) / 35)
```

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];
```



If this accuracy is not good enough, simply choose a higher order Padé approximant:

```
In[7]:= dawson78[x_] = Pade[dawson[x], {x, 0, 7, 8}];
```

```
In[8]:= Table[dawson[x] - dawson78[x] // N, {x, -1, 1, 0.2}]
```

```
Out[8]= {-5.28332 10^-8, -1.71052 10^-9, -1.71491 10^-11,
-2.1445 10^-14, -1.49078 10^-19, 4.84891 10^-38,
1.76183 10^-19, 2.14451 10^-14, 1.71491 10^-11,
1.71052 10^-9, 5.28332 10^-8}
```

Thus, dawson78 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[9]:= Horner[p_?PolynomialQ, x_] :=
Fold[x #1 + #2 &, 0,
Reverse[CoefficientList[p, x]]]
```

Applying Horner to dawson78, one obtains:

```
In[10]:= (Horner[Numerator[#, x], x] /
Horner[Denominator[#, x], x]) & @ dawson78[x]
```

```
Out[10]= (x (1 + x^2 (-2/15 + x^2 (4/117 - 248 x^2 / 225225))) /
(1 + x^2 (8/15 + x^2 (8/65 + x^2 (-32/2145 + 16 x^2 / 19305))))
```

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

```
In[11]:= FortranForm[N[%]]
Out[11]//FortranForm=
x*(1. + x**2*(-0.13333333333333333333 +
- x**2*(0.03418803418803418803 -
- 0.001101121101121101121*x**2)))/
- (1. + x**2*(0.5333333333333333333 +
- x**2*(0.1230769230769230769 +
- x**2*(0.01491841491841491841 +
- 0.0008288008288008288009*x**2))))
```

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

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

```
In[1]:= << Algebra`SymbolicSum`
```

it is straightforward to find a closed form for f :

```
In[2]:= Sum[1/((x+k)^2 + 1), {k, -Infinity, Infinity}]
Out[2]=  $\frac{I}{2} \text{Pi Cot}[\text{Pi} (1 + I - x)] + \frac{I}{2} \text{Pi Cot}[\text{Pi} (-1 + I + x)]$ 
```

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

```
In[3]:= ComplexExpand[%]
Out[3]=  $I \left( \frac{-(\text{Pi Sin}[2 \text{Pi} (1 - x)])}{2 (\text{Cos}[2 \text{Pi} (1 - x)] - \text{Cosh}[2 \text{Pi}])} - \frac{\text{Pi Sin}[2 \text{Pi} (-1 + x)]}{2 (\text{Cos}[2 \text{Pi} (-1 + x)] - \text{Cosh}[2 \text{Pi}])} \right) - \frac{\text{Pi Sinh}[2 \text{Pi}]}{2 (\text{Cos}[2 \text{Pi} (1 - x)] - \text{Cosh}[2 \text{Pi}])} - \frac{\text{Pi Sinh}[2 \text{Pi}]}{2 (\text{Cos}[2 \text{Pi} (-1 + x)] - \text{Cosh}[2 \text{Pi}])}$ 
```

But expanding the trig functions using

```
In[4]:= Expand[%, Trig -> True] // ExpandAll
```

```
Out[4]=  $-\frac{\text{Pi Sinh}[2 \text{Pi}]}{\text{Cos}[2 \text{Pi} x] - \text{Cosh}[2 \text{Pi}]}$ 
```

yields an explicitly real closed form for $f(x)$. The expected periodicity is evident through the `Cos[2 Pi x]` term.

Testing Pattern Matching

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

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

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

```
In[2]:= Alias["_"]
Out[2]= Blank
```

is replaced by

```
In[3]:= Alias["__"]
Out[3]= BlankSequence
```

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

```
In[4]:= {a, b, c} /. {x___, y_} :> Anything /;
Print["Trying ", "x -> ", {x}, ", ", "y -> ", {y}]
Trying x -> {a, b}, y -> {c}
Out[4]= {a, b, c}
```

The general pattern involving

```
In[5]:= Alias["___"]
Out[5]= BlankNullSequence
```

takes into account all possible matches:

```
In[6]:= {a, b, c} /. {x___, y___} :> Anything /;
Print["Trying ", "x -> ", {x}, ", ", "y -> ", {y}]

Trying x -> {}, y -> {a, b, c}
Trying x -> {a}, y -> {b, c}
Trying x -> {a, b}, y -> {c}
Trying x -> {a, b, c}, y -> {}
Out[6]= {a, b, c}
```