Getting Started with ΔF Strings for Formatting

Description

ΔF is a utility for Dyalog APL that performs a function similar to string formatters in other languages, such as Python's f-strings, but with an *APL* flair.

Outline

Setup

Let's get started!

Let's be sure the file **ΔFormat.dyalog** is accessible and loaded. When fixed, it creates a single function `ΔF`.

In [1]:
```
⍝ We can start the Jupyter _ipynb_ file up in the same directory as ΔFormat.dyalog.
'1. Our active directory is ',⎕SH 'pwd'
'2. Loading ΔFormat.dyalog...'
2 ⎕FIX 'file://ΔFormat.dyalog'
'3. Whoops!' '3. ΔF Exists!'⊃⍨3=⎕NC 'ΔF'
```

Out[1]:
```
1. Our active directory is  /Users/petermsiegel/MyDyalogLibrary/pmsLibrary/src
```

Out[1]:
```
2. Loading ΔFormat.dyalog...
```

Out[1]:
```
3. ΔF Exists!
```

Before showing how ΔF works, let's display a couple of variables...
Here are the variables...

In [2]:
```
string  ← 'This is a string'
numbers ← 10 20
```

We will display them using ΔF.

In [3]:
```
ΔF 'My string = "{string}".  My numbers = {numbers}.'
```

Out[3]:
```
My string = "This is a string".  My numbers = 10 20.
```

Or, more concisely using Self-documenting Code Fields, which we'll discuss below.
(Note: The symbol ➤ is a special right arrow that delimits the literal code from its value.)

In [4]:
```
ΔF '{string → }.  {numbers → }'
```

Out[4]:
```
string ➤ This is a string.  numbers ➤ 10 20
```

But, let's start at the **beginning**!

ΔF Fields

Text Fields: `'A simple string'`

The simplest possible format string-- we'll call it an **ΔF** string-- consists of a simple **Text** field.

In [5]:
```
ΔF 'This is a simple string.'
```

Out[5]:
```
This is a simple string.
```

Text Fields: Newlines and Escapes

Text fields can consist of one or more lines, each separated by the special newline escape sequence `\⋄` . Using **Text** fields this way is just one way to create a list of items or a multiline paragraph.

In [6]:
```
ΔF 'This\⋄is a\⋄multiline string.'
```

Out[6]:
```
This
is a
multiline string.
```

You can insert most any Unicode character into a **Text** field. Only four characters (with special meaning *described below*) require special treatment:

`{` , `}` , `⋄` , and `\` .

A diamond `⋄` *not* preceded by a backslash `\` has no special meaning; only the escaped sequence `\⋄` denotes a newline. You'll see below that a bare `{` begins a **Code** field, which is terminated by a balancing bare `}` . Finally, literal backslashes only need to be escaped themselves (i.e. as `\\`) when right before a `⋄` , `{` , `}` , or another backslash `\` . In other cases, such as `+\⍳` no extra backslash is required.

In [7]:
```
ΔF '"We can sum the numbers ⍵ via +\⍵"'
```

Out[7]:
```
"We can sum the numbers ⍵ via +\⍵"
```

Code Fields: Simple Variables `'{lastName}, {firstName}'`

Let's create a more useful example of ΔF strings using the following three variables.

In [8]:
```
what← 'This'
type← 'simple'
thing← 'string'
```

Within separate sets of curly braces `{..}` , which delimit a **Code** field, we include the three variable names: `what` , `type` , and `thing` . We'll say more about **Code** fields in a moment.

In [9]:
```
ΔF '{what} is a {type} {thing}.'
```

Out[9]:
```
This is a simple string.
```

Knowing Your Fields

This ΔF string consists of six fields:

1. a **Code** field `{what}` , which returns the value of the variable `what` ;
2. a **Text** field `" is a "` ;
3. another **Code** field `{type}` , returning the value of the variable `type` ;
4. a short **Text** field `" "` ;
5. a **Code** field `{thing}` , referencing `thing` ; and finally,
6. a final **Text** field `"."` .

Debug Mode: `'debug'`

We can show each of the fields more graphically using the *debug* option (abbreviated *d*), which places each field in a separate display box and marks each space in each field by a middle dot `·` .

In [10]:
```
'd' ΔF '{what} is a {type} {thing}.'
```

Out[10]:
```
┌→──┐┌→─────┐┌→─────┐┌→┐┌→─────┐┌→┐
↓This│↓·is·a·│↓simple│↓·│↓string│↓.│
└───┘└──────┘└──────┘└─┘└──────┘└─┘
```

Code Fields Are DFNS: `'{⍪⍳3} {↑"Name" "Addr" "Phone"}'`

As shown above, in addition to **Text** fields, we can create executable **Code** fields, using braces {...}. A **Code** field with a bare variable names is the simplest type of **Code** field.

Code fields can be generalized as dfns* evaluated in the active (caller's) namespace. While each **Code** field is executed via ordinary APL rules (statements left-to-right and right-to-left within statements), **Code** fields within a `⎕FMT` format string are themselves executed left-to-right:

```
the left-most Code field is executed first, then the one to its right, and so on.
```

Each **Code** field* *must* return a value (perhaps a null string).

```
+-------------------------------------------------------------------------+
| * A Code field may end with a comment, which starts with a lamp `⍝`     |
|   and contain no braces or ⋄. Example:  { ?0  ⍝ A random number ⍵: 0<⍵<1 } |
+-------------------------------------------------------------------------+
```

Let's look at more complex examples. First, what if a variable itself is more than a simple one-line text string?

In [11]:
```
nums←  ⍳3
what← ↑'This' 'That' 'The other thing'
type← ↑'simple' 'hard' 'confusing'
thing←↑'string' 'matrix' 'thingamabob'
⎕FMT '{nums} {what} is a {type} {thing}'
```

Out[11]:
```
0 This            is a simple    string
1 That                 hard      matrix
2 The other thing      confusing thingamabob
```

Here, `num` is a column vector of integers, and `what`, `type` and `thing` are character matrices. Any object that can be formatted via Dyalog `⎕FMT` can be returned from a **Code** field.

Now for a more complex example. You can place arbitrary APL code within the braces `{...}` of a **Code** field.

In the example below, we'll remove the ↑ prefix from the values of each of these three variables. Notice how we insert a period after each word of the variable `thing` and create a quoted string using double quotes: `{ ↑thing,¨"." }` Such a string is called a **DQ String** and appears only within **Code** fields.

In [12]:
```
what←'This' 'That' 'The other thing'
type←'simple' 'hard' 'confusing'
thing←'string' 'matrix' 'thingamabob'
⎕FMT '{ ⍳≠what } { ↑what } is a { ↑type } { ↑thing,¨"." }'    ⍝ See below regarding "."
```

Out[12]:
```
0 This            is a simple    string.
1 That                 hard      matrix.
2 The other thing      confusing thingamabob.
```

DQ Strings in Code Fields: Use `"These"` *not* `'These'`

Within **Code** fields, strings *require* double quotes (`"`). These **DQ strings** `"like this one"` are used wherever single-quoted strings `'like this'` would be used in standard APL; single-quoted strings are ***not*** used. Single quotes may appear, most usually as literal characters, rather than to create strings.

DQ Strings: Escapes `\⋄`

DQ Strings support the escaped sequences `\⋄` and `\\`. `\⋄` is a convenient way to enter newlines (actually ⎕UCS 13, the carriage return character) into linear strings. When the `⎕FMT` *string* is printed, newlines will create separate lines in the output matrix.

To include an actual double quote within a **DQ String**, double the doublequote (`""`), just as one would do for single quotes in standard APL strings. Single quotes are doubled on entry as required by APL when entering the ⎕FMT format string. Notice how the string below is a 3-row matrix, one row for each line of the **DQ String**.

In [13]:
```
⍝                  Row 1 Row 2 Row 3...
'# rows:',≠⊢← ⎕FMT '{"This\⋄is a\⋄""DQ"" field, isn''t it?"}'
```

Out[13]:
```
This
is a
"DQ" field, isn't it?
# rows: 3
```

Space Fields*: { }

The third and last field type is a **Space** field, which looks just like a **Code** field, except that it contains only zero or more spaces between the braces { }*. *A space field forms a separate field and is a good way to separate Text fields.*

```
+-------------------------------------------------------------------------+
| * A space fields may include a comment, which starts with a lamp `⍝`    |
```

```
|   and contains no braces or ⋄. Example:   {   ⍝ 3 spaces}                    |
+-----------------------------------------------------------------------------+
```

In [14]:
```
      ∆F 'This is{ }a test.'
'd' ∆F 'This is{ }a test.'
```

Out[14]:
```
This is a test.
```

Out[14]:
```
┌→──────┐ ┌→┐ ┌→──────┐
↓This·is│ ↓·│ │↓a·test.│
└───────┘ └─┘ └───────┘
```

But why bother with space fields?

- They are useful when separating out multiline string or code fields; even a zero-width space field can separate two **Text** fields; and
- They ensure the expected amount of spacing when preceded or followed by text fields with lines of varying length.

Here's an example of two multiline **Text** field separated by a **Space** field with a single space: { }.

In [15]:
```
      ∆F 'This\⋄is a\⋄multiline\⋄field!{ ⍝ 1 Space}{"This\⋄is\⋄as well!"}'
```

Out[15]:
```
This      This
is a      is
multiline as well!
field!
```

In [16]:
```
'd' ∆F 'This\⋄is a\⋄multiline\⋄field!{ ⍝ 1 Space}{"This\⋄is\⋄as well!"}'
```

Out[16]:
```
┌→──────────┐ ┌→┐ ┌→──────────┐
↓This·····│ ↓·│ │↓This····│
│is·a·····│ └─┘ │is·····│
│multiline│     │as·well!│
│field!···│     └────────┘
└─────────┘
```

In this next example, we use a zero-width **Space** field simply to allow us to create two independent **Text** fields:

In [17]:
```
      ∆F '1. \⋄2.\⋄3.{}Jane\⋄John\⋄Nancy'   ⍝ Or equivalent:  '1.\⋄2.\⋄3.{ }Jane\⋄John\⋄Nancy'
```

Out[17]:
```
1. Jane
2. John
3. Nancy
```

Pseudo-builtin $

Here's how to do this more elegantly using the pseudo-builtin `$`, which is a nice way to use the Dyalog APL formatting utility ⎕FMT.

In [18]:
```
      ∆F '{"I1,⊂. ⊃" $ 1+⍳3}Jane\⋄John\⋄Nancy'
```

Out[18]:
```
1. Jane
2. John
3. Nancy
```

Now, let's move on to a few more examples.

In [19]:
```
      ∆F 'Multiples of pi: {"I1,⊂×Π =⊃" $ 1+⍳4} {"F10.7" $ ○1 2 3 4}'
```

Out[19]:
```
Multiples of pi: 1×Π =  3.1415927
                 2×Π =  6.2831853
                 3×Π =  9.4247780
                 4×Π = 12.5663706
```

Again, using the *debug* option, we can see exactly what fields are set up.

In [20]:
```
'd' ∆F 'Multiples of pi: {"I1,⊂×Π =⊃" $ 1+⍳4} {"F10.7" $ ○1 2 3 4}'
```

Out[20]:
```
┌→──────────────────┐ ┌→────┐ ┌→┐ ┌→──────────┐
↓Multiples·of·pi:·│ │↓1×Π·=│ ↓·│ │↓·3.1415927│
└───────────────────┘ │2×Π·=│ └─┘ │·6.2831853│
                      │3×Π·=│     │·9.4247780│
                      │4×Π·=│     │12.5663706│
                      └─────┘     └──────────┘
```

Pseudo-builtin Function $$ for a boxed display

If we want a **Code** field to be boxed in the regular output, we can use the pseudo-builtin display function **$$**. Using **$$**, no middle dots (·) appear, unless you create them yourself!

In [21]:
```
      ΔF 'Multiples of pi: {$$ "I1,⊂×Π =⊃" $ 1+⍳4} {$$ "F10.7" $ ⍳1 2 3 4}'
```

Out[21]:
```
Multiples of pi: ┌→────┐ ┌→──────────┐
                 ↓1×Π =│ ↓ 3.1415927│
                 │2×Π =│ │ 6.2831853│
                 │3×Π =│ │ 9.4247780│
                 │4×Π =│ │12.5663706│
                 └─────┘ └──────────┘
```

ΔF arguments with ⍵0 ... ⍵99 and ⍵ (or ω0 ... ω99 and ω_)

ΔF supports the use of arguments to **ΔF**, including the ΔF format string itself. The format string is designated ⍵0, and each subsequent argument is ⍵1, ⍵2, etc. These designations can be used in place of ((⍳⎕IO)⊃⍵), et cetera, within **Code** fields:

In [22]:
```
      ΔF '{⍵1} multiples of pi: {"I1,⊂×Π =⊃" $ 1+⍳⍵1 } {"F10.7" $ ⍳ 1+⍳⍵1}'  3
      ΔF '{⍵1} multiples of pi: {"I1,⊂×Π =⊃" $ 1+⍳⍵1 } {"F10.7" $ ⍳ 1+⍳⍵1}'  2
```

Out[22]:
```
3 multiples of pi: 1×Π =  3.1415927
                   2×Π =  6.2831853
                   3×Π =  9.4247780
2 multiples of pi: 1×Π =  3.1415927
                   2×Π =  6.2831853
```

The symbol ΔF ⍵ alone will select the **next** argument in sequence (one past the **current** argument, which is the last one selected directly, e.g. ⍵5, via ⍵, or ⍵0 if the first use). This makes it easy to format a set of items:

In [23]:
```
      ΔF 'Rate: {⍵ $ ⍵}; cur. value: {⍵ $ ⍵}' 'F5.2,⊂%⊃' (2.200 3.834 5.996) '⊂£⊃,F7.2' (1000.23, 2250.19 2500.868)
```

Out[23]:
```
Rate:  2.20%; cur. value: £1000.23
       3.83%              £2250.19
       6.00%              £2500.87
```

Note: You may enter ω0, ω1, etc. as equivalents to ⍵0, ⍵1, etc. and ω_ for ⍵ alone. Note also that ⍵0 can never be selected via the lone ⍵, since the *last* index specified is never less than 0, so the *next* is never less than 1.

Justification and Centering Codes *L*, *C*, *R* with Pseudo-function $

The pseudo-function $ has been extended with 3 special codes for left-justified Lnn, centered Cnn, and right-justified Rnn output within a **Code** field. This is valid for both numeric and text data. Only one special code may be used in each $ call (but you may call $ itself more than once) and that code must be the *first* or *only* code specified. If other (usually numerically-oriented) codes follow, a comma must intervene (following the style of dyadic ⎕FMT).

Here, we *left-, center-*, and *right*-justify Names in the ΔF arguments.

In [24]:
```
      names←↑'John' 'Mary'
      ΔF '<{"L10" $ ⍵1}> <{"C10" $ ⍵1}> <{"R10" $ ⍵1}>' names
```

Out[24]:
```
<John       > <   John    > <       John>
 Mary            Mary            Mary
```

In [25]:
```
      'd'  ΔF '<{"L10" $ ⍵1}> <{"C10" $ ⍵1}> <{"R10" $ ⍵1}>' names
```

Out[25]:
```
┌→┐┌→──────────┐┌→──┐┌→──────────┐┌→──┐┌→──────────┐┌→┐
↓<│↓John·····│↓>·<│↓···John···│↓>·<│↓·····John│↓>│
└─┘│Mary·····│└───┘│···Mary···│└───┘│·····Mary│└─┘
   └──────────┘     └──────────┘     └──────────┘
```

Justification and Centering Codes , *l*, *c*, *r* with Pseudo-function $: Vector args as *1-Row Matrix* **vs** *Column Vector*

Like standard ⎕FMT, $ *by default* considers simple vectors in the code field as column vectors (as in the example above). This is true even for the extensions L, C, and R. However, you can override this, by specifying justification codes in lower case: l, c, or r. If these are used, simple vectors in the code field used as arguments to $ are treated as 1-row matrices instead.

- Right arguments that are not simple or are not a numeric or character vector are not impacted.

Here, `"c0"` (or `"l0"` or `"r0"`) formatting with `$` ensures a simple vector right argument (numeric or character) is treated as a 1-row matrix. Similarly, `"C0"` (or `"L0"` or `"R0"`) formatting with `$` ensures a simple vector right argument (numeric or character) is treated as a column vector, even if no standard ⎕FMT codes are used.

- The code `"c0"` or `"C0"` works because justification and centering codes ensure a **minimum** width, never truncating.

In [26]:
```
      ΔF 'For n∈1 2 3, nΠ = { "c0" $ ○ ω1 }.   {"I1,⊂Π = ⊃" $ ω1} { "C0" $ ○ω1 }' (1 2 3)
```

Out[26]:
```
For n∈1 2 3, nΠ = 3.141592654 6.283185307 9.424777961.   1Π =  3.141592654
                                                         2Π =  6.283185307
                                                         3Π =  9.424777961
```

Miscellaneous Options

Code field namespace α

For **Code** fields, ΔF passes a namespace as the left argument (α). That namespace contains all the support functions and variables for ΔF. Names beginning with an underscore `_` (e.g. `_`, `__`, `_myVar`, ...) are reserved for the user's use. One potential use is setting state that is maintained across all **Code** fields during the execution of ΔF, without cluttering the calling environment:

In [27]:
```
      ΔF '{α._PITimes←(○1)∘× ⋄ α._PITimes 1} {α._PITimes 2}'
```

Out[27]:
```
3.141592654 6.283185307
```

In [28]:
```
      ⎕FR←645 ⋄ ⎕PP ←10
      ΔF 'To 34 digits, Pi={α._sav←⎕FR ⎕PP  ⋄ ⎕FR ⎕PP∘←1287 34  ⋄ (⎕FR ⎕PP∘←α._sav)⊢$ ○1 }'
      ('⎕FR still 645? ',(⎕FR≡645)⊃'No.' 'Yes.')'    ' ('⎕PP still  10? ',(⎕PP≡10) ⊃'No.' 'Yes.')
```

Out[28]:
```
To 34 digits, Pi=3.1415926535897932384626433832795 03
```

Out[28]:
```
 ⎕FR still 645? Yes.       ⎕PP still  10? Yes.
```

ΔF for assertions.

Normally, ΔF returns the formatted text as a single formatted matrix (rank 2).
If the left argument (α) to ΔF is a homogeneous numeric array, it is viewed as an assertion.

- If the assertion contains *no numeric zeroes*, it is **true**. It **prints** the formatted text, returning a shy *1*. (It does *not* return the formatted text, as in *format* mode.)
- If the assertion contains one or more zeroes, it is **false**. It does nothing, returning a shy *0*.

In [29]:
```
      ⍝ Here, var is in range, so no ΔF string message is produced. 0 is returned.
      var←100
      ⎕←(var<100) ΔF 'Warning! Variable "var" is out of range: var={var}'
```

Out[29]:
```
0
```

In [30]:
```
      ⍝ Now, var is out of range (the assertion is true), so a ΔF string message is printed. 1 is returned.
      var-←1
      ⎕←(var<100) ΔF 'Warning! Variable "var" is out of range: var={var}'
```

Out[30]:
```
Warning! Variable "var" is out of range: var=99
1
```

ΔF Syntax

```
   Syntax:  [assertion | options] ΔF format_string [arbitrary args]

     α/assertion: α must be a simple numeric array. "TRUE" unless there is a 0 in the left arg.
        ∘ If TRUE, prints formatted result returning shy 1. Otherwise, does nothing, returning shy 0.

     α/options: DEBUG | COMPILE | HELP | DEFAULT*
        ∘ DEBUG: Displays each field separately using dfns "DISPLAY"
          The abbrev 'DE' or 'D' denotes DEBUG.
        ∘ COMPILE: Returns a code string that can be converted to a dfn (executed via ⍎),
           rather than scanned on each execution.
          – The resulting dfn should have a dummy format ''. If a non-empty string, that
            is treated as if the format string ω0.
          – If the resulting dfn is called with a left arg α,
            it must be an assertion (numeric array) and handled as above.
        ∘ HELP: Displays HELP documentation (ω ignored):
            ΔF≈'HELP'
        ∘ DEFAULT: Returns a formatted 2-D array according to the format_string specified.
```

```
     * DEFAULT is assumed if α is omitted or ''. Options may be in either case and abbreviated.

format_string (ω[0]):
   ◦ Contains the simple format "fields" that include strings (text fields), code (code fields), and
     2-D spacing (space fields). Code fields accommodate a shorthand using
     ◦ $ to do numeric formatting (via ⎕FMT) and justification and centering, as well as
     ◦ $$ to display fields or objects using dfns 'DISPLAY'.
arbitrary args (1↓ω):
   ◦ Optional arguments to be referenced in Code Fields.
   ◦ In place of argument references: (0⊃ω), (1⊃ω)  or better (ω⊃⍨0+⎕IO), (ω⊃⍨1+⎕IO)
     arguments can be referred to in Code Fields via
           ω̲0, ω̲1, ..., or a bare ω̲,  or alternatively
           ω0, ω1, ..., or a bare ω_.
     ω̲1 or ω1 is the first APL argument after the format string.
     ω̲10 or ω10 is the 10th such arg.
     ω̲0 or ω0 refers to the format string itself.
     ω̲ or ω_ refers to the next arg measuring from left to right across all Code Fields.
     If no prior ω̲ specification appears,
        ω̲ or ω_ refers to ω̲1.
     If the previous reference (left to right) was ω̲10, ω̲ refers to ω̲11.
        Example: {ω̲ ω̲5 ω̲ ω̲1}  {ω̲ ω̲10 ω̲}  is the same as  {ω̲1 ω̲5 ω̲6 ω̲1} {ω̲2 ω̲10 ω̲11}.
```