Format Strings {A} ∆format B [C0 C1 …]

{A} ∆f B [C0 C1 …]

Format strings provide a simple way to display a mixture of text, APL variables, and arbitrary expressions in a straightforward way, including (where required) precise formatting of native multidimensional and multi-line objects.[1] They are similar to **f-strings** in python[2] and similar constructions in other languages, but designed for handing APL arrays.

Preview

Before we get started, we'll share two examples to give you a sense of what format strings are all about.

Example 1: Here is a most simplistic example to give you a flavor of format strings:

```
   time←'night'  ◊  who1←'we' ◊  where1←'bed'
   who2←'Mrs O''Leary' ◊ what←'lantern' ◊ where2←'shed'
 ⍝ Here, we use the variables time, who1, etc. within the format string
   ∆f 'One dark {time}, when {who1} were all in {where1},⍎↓{who2} lit a {what}
                                                   in the {where2}!'
One dark night, when we were all in bed,
Mrs O'Leary lit a lantern in the shed!
```

Example 2: And here is an example that uses some more of the power of ∆**f** and APL:

```
   Names←'John Jones' 'Mary Smith' 'Jan Ito'
   Salaries←125000 132000 85000
 ⍝ Here we apply APL ↑ to Names,
 ⍝ and use a shortcut to APL ⎕FMT specs to format Salaries.
   ∆f'Name{8}Salary⍎↓{↑Names}  {⊂£⊃,G⊂ZZ9,999⊃$Salaries}'
Name        Salary
John Jones  £125,000
Mary Smith  £132,000
Jan Ito     £ 85,000
```

Arguments

{A} (Options):

A=1 <u>default</u>: ∆**format or** ∆**f** returns a formatted version of its right-argument string. If a single line, a character vector is returned; if more than one line, a character matrix is returned. (If multidimensional objects are formatted, they are mapped into the matrix result, with appropriate spacing, just as for **⎕FMT**).

A=0: ∆**f** returns a parsed string of pseudo-code that shows exactly how the right argument was interpreted. The pseudo-code is normally not executable, but useful for debugging.

A=2: Echos the ∆**f** command and argument string (**C**) in a canonical form:

∆format '*string*' ['*pos1*' '*pos2*' ...]

---

[1] In this documentation, we'll use the term ***format string***, with **∆f** standing in for the synonyms, ∆**format** and ∆**f.** Several field types, including **Code Fields**— **{}**-format strings— will be demonstrated. Name and Quote Fields, which begin with ⍎, have similar options, but may not appear in the final release of ∆**f** (see Appendix A).

⎕IO ⎕ML←0 1 will be used throughout.

Help Info: For help information within the **∆f command,** enter: **∆f '⍰?'**

For typical initialization of format functions, see **Appendix A** (below).

[2] Format strings here are not based on any particular language's constructions.

B (Format String):
The primary right argument to ∆**f** is a single character vector (after any evaluation, catenation, etc.), including both literal text along with special characters like **{}**, **⍎, $**, and ⎕, to indicate APL variables and code to evaluate, along with specifications for formatting their output.

C (Positional and Immediate Variables).:
The primary right argument, **B**, may contain references to external objects by name or through positional objects passed to the ∆**f** function, directly after the **format string.** Each of these positional values **C0, C1, …,** is handled in one of two ways, depending on whether it is of the form **⍵*nnn*** or **⍺*nnn,*** where ***nnn*** is an integer such that 0 refers to the first item *after* the format string, 1 to the second, and so on.

Positional Variables are denoted by **⍵0, ⍵1, …,** which are treated like any named APL variable referenced within the format string and may be of any expected type**; a**nd

Immediate Variables are denoted by **⍺0, ⍺1, …,** whose text is immediately substituted into the format string *before* any other processing; these must be convertible to character vectors (or scalars), via **⍕⍺0**, etc. These are useful for setting headers, footers, and formatting specifications based, for example, on data with changing length or type.

If a positional or immediate variable does not refer to an actual argument to ∆**f**, its value defaults to its name (e.g. **⍵1** by default has '**⍵1**' as its literal value**).**

Result:
∆**f** by default returns the formatted expression that results from evaluating the expressions and specifications in **B**. If valid, that expression is *guaranteed* to be a simple character vector, if the result fits on a single APL line; otherwise, the expression is a character matrix.

Specifications within B (the format string)

Fields: **Format Strings** can be divided into **fields**. Each **field** will be formatted as a 2-dimensional, rectangular object, one catenated to the next, left to right.

Literal Fields: Simple text that doesn't include (unescaped) characters with special meanings— **{}**, **⍎**, **⎕**, **⌹**, **$,** or quotes (**'** and **"**)— may be typed in as is. A string may consist of multiple lines, with ⍎◊ signifying a new line.

```
      ∆f 'This is an example⍎◊of two lines.'
This is an example
Of two lines.
```

Code Fields: Code to be evaluated at run-time is entered within braces **{...}.** Code may include any standard APL, including dfns, with some extensions described below; double-quoted strings will be recognized (making string constants easy to enter) and appropriately converted to single-quoted strings on execution. Finally, **Code Fields** may not consist of a single unadorned integer (surrounded by 0 or more spaces), e.g. **{15}**; that syntax is reserved for **Space Fields**, below.

In this example, we have three independent fields, the **Literal Field** on the left ends when the **Code Field** (with its left brace) starts, and the **Literal Field** on the right begins, when the **Code Field** ends (with the right brace).

```
      ∆f 'One⍎◊two⍎◊three {↑"cat" "dog" "mouse"} four⍎◊five'
One   cat   four
two   dog   five
three mouse
```

Note how the first "field," consisting of "One...three" is self-contained vertically, as are the other two fields, "{...}" and " four...five". {} alone represents a null field, whose only function is to separate simple text fields or other fields:

```
      ∆f 'One⍎◊two⍎◊three{bar}cat⍎◊dog⍎◊mouse{bar←↑3⍴⊂" | "}four⍎◊five'
One   | cat   | four
two   | dog   | five
three | mouse |
```

Null/Space Fields: ∆**f** also supports space fields. The sequence {nnn} will insert an nnn-space wide field, where nnn is a simple non-negative integer:

```
      ∆f 'One⍎◊two⍎◊three{3}cat⍎◊dog⍎◊mouse{2}four⍎◊five'
One     cat    four
two     dog    five
three   mouse
```

Here, the left-most field is separated from the middle field by 3 spaces, which is separated in turn from the right-most field by 2 spaces. To evaluate code that consists of a single integer, enter anything in the field besides an integer,[3] e.g. **{2.0}, {,2}, {⊢2}**, etc.

A pair of single literal braces may be entered into a string by doubling to `{{…}}`.

```
      ∆f 'This is {{a}} test.'
This is {a} test.
```

A 0-length space field **{0}** or its variant **{}** can be used to separate each rectangular (2-dimensional) field from the next, without adding more spacing [Note: our next example will do this more elegantly]:

```
      ∆f '1⍎◊2⍎◊3{}4⍎◊5⍎◊6'
14
25
36
```

When bare braces are used like this **{}** to conclude one field before beginning another, we call that a Null Field, equivalent to **{0}**, a zero-length Space Field.

```
      ∆f '{⍪1+⍳3}{⍪4+⍳3}'      ⍝ Same output as above!
```

Unicode and Numeric pseudo-variables: Literal braces, { and }, may also be entered as **⎕UCS** pseudo-variables, **⎕U123** and **⎕U125**, or **⎕U7BX** and **⎕U7DX**:

---

[3] Extra spaces are also ignored within {nnn} fields, i.e. { 5 } is the same as {5}. Note that negative integers like **{¯3}** are invalid in Space Fields; use a literal field or add other code: **{⊢¯3}.**

```
      ∆f 'This is ⎕U123a⎕U125 ⎕U7BXthe⎕U7DX test.'
This is {a} {the} test.
```

More generally, any Unicode character may be entered via decimal **⎕U*ddd*** or hexadecimal **⎕U*dhh*X** (or **⎕U*dhh*x**), where ***d*** is in **[0-9]** and ***h*** in [**0-9a-fA-F]**.[4] **⎕N*ddd*** specifies an integer in decimal which is displayed as *hexadecimal*, and **⎕N*dhh*** specifies hexadecimal to decimal:

```
      ∆f 'α0 in hex is ⎕Nα0. And α1x in decimal is ⎕Nα1X'  45 '2D'
45 in hex is 2D And 2Dx in decimal is 45
```

Code Expressions: The general form for a code field is:

```
      {[[fmt1|fmt2]+ '$'] code}   or { [[fmt1|fmt2]+ '$$'] code}
```

Where ***fmt1*** consists of the standard formatting codes of APL **⎕FMT** without quotes and ***fmt2*** consists of special *extensions* to APL **⎕FMT** for left- and right-justification and centering. The latter may be used with *any* APL extension, but the former must be within **⎕FMT**'s domain. Let's start with **fmt1, ⎕FMT-code expressions:**

⎕FMT-Code Expressions[5]:

```
      ∆f '{I4$1+⍳3}{I3,⎕ <⎕,I2,⎕> ⎕,I2$3 3⍴4+⍳9}'
2  5 < 6>  7
3  8 < 9> 10
4 11 <12> 13
```

Here, the single dollar-sign ($) both delimits formatting from code, while denoting that the code is treated according to **⎕FMT** rules[6], whereby a vector right argument, here **2 3 4,** is treated as a one-column matrix, as if:

```
      'I4' ⎕FMT 2 3 4
```

The expression **3 3⍴4+⍳9** is acted on by **⎕FMT** specifications '**I3,⎕ <⎕,I2,⎕> ⎕,I2**'; that is, a 3-digit integer, the text '**<**', a 2-digit integer, the text '**>** ', and finally a 2-digit integer.

The double dollar sign ($$) functions identically to the single dollar sign ($), *except* that a right-hand vector will be treated as if a one-row matrix, contrary to **⎕FMT** defaults, rather than as a one-column vector.

```
     ∆f '↓ {I4$1+⍳3} → {I4$$1+⍳3}'
↓ 2   →  2   3   4
  3
  4
```

---

[4] To escape **⎕U*ddd*** or **⎕U*hhh*X**, use **⎕⎕U...: ⎕⎕U123** appears as the literal string **⎕U123.**
[5] For delimiters, use those valid for **⎕FMT**, i.e. : **⊂ ⊃, ¨ ¨, ⎕ ⎕, < >, or ⎕ ⎕.**
[6] APL expressions ***not*** within the scope of a dollar-sign ($) follow normal APL conventions, , i.e. not those of **⎕FMT**

⎕FMT-Code Extensions: There are currently three additional formatting expressions within Code Fields, which affect the evaluated code expression ***code*** to the right of the $ (or $$):

C*nnn*	C*nnn*⊂*c*⊃	Center ***code*** in a field *nnn* characters wide.
L*nnn*	L*nnn*⊂*c*⊃	Left-justify ***code*** in a field *nnn* characters wide.
R*nnn*	R*nnn*⊂*c*⊃	Right-justify ***code*** in a field *nnn* characters wide.

When ***⊂c⊃*** is specified, a single character ***c*** will be used to pad the field; it may be specified as follows:

- A literal character, including a single digit, ***except*** an enclosing delimiter, a single or double quote, parenthesis, or brace (see 2nd and 3rd options[7]);
- A number *nn* of 2 ***or more*** digits, where ***nn***>32, replaced on output by **⎕UCS *nn***. Any value out of range will be treated as the **RC** (Unicode replacement character) below.
- Any of these 2-character names:

SQ single quote	**'**	**LP** left parenthesis	**(**
DQ double quote	**“**	**RP** right parenthesis	**)**
MD middle dot (⎕U183)	**·**	**LB** left brace	**{**
SP space	**' '**	**RB** right brace	**}**
KS Kanji (wide) space (⎕U12288)	'　'		
RC replacement character (⎕U65533)	�		

Here’s an example:

```
 str ←'{L20⊂MD⊃,I7$10*1 2 3 4 5} '
str,←'{C10⊂→⊃$↑"Mon" "" "Wed" "" "Fri"}'
str,←'{R20⊂183⊃,F5.2,X1$?5⍴0}' ⍝ (⎕UCS 183) ≡ Middle Dot
∆f str
    10·············· →→→→Mon→→→·············· 0.95
   100·············· →→→→   →→→·············· 0.67
  1000·············· →→→→Wed→→→·············· 0.06
 10000·············· →→→→   →→→·············· 0.56
100000·············· →→→→Fri→→→·············· 0.91
```

Miscellaneous Features: Here we describe several useful advanced features before diving into the fundamentals of advanced features of f-strings.

Headers and Footers: There F-strings allow for (multi-line) headers and footers, that are centered above or below the entire formatted object. The basic format uses the ***option prefix*** **⌹,** a letter **[H|F],** and a **⎕FMT**-style quoted specification:

⌹H⊂*header*⊃ ⌹F⊂*footer*⊃

All special symbol constructors like **⎕U*nnn*, ⍎◊,** and so on are available. In this example, we add a single-line header and a 2-line footer.

---

[7] This is a limitation (and workaround) due to how format strings are scanned for **‘(){}”** in the prototype.

```
⍝ Student Grades #1
    Names←'John Jones' 'Mary Smith' 'Fran Allen' 'Ted Chu'
    Scores←97 85 92 55 ◊ Grades← 'A' 'B' 'A-' '*'
    Hdr←'⎕H⊂Student Grades⊃'
    Ftr←'⎕F<_____________⍎◊* Incomplete>'
    ∆f Hdr,Ftr,'{↑Names}  {⍕Scores}  {↑Grades}'
  Student Grades
John Jones  97  A
Mary Smith  85  B
Fran Allen  92  A-
Ted Chu     55  *
  _____________
  * Incomplete
```

`Positional ⍵n and Immediate Variables ⍺n`: In the example above, the format statement got long, so we catenated the header and footer strings on the **∆f** command line. An alternative is to use an ***immediate variable.*** Immediate variables are of the form **α0** to **α9,** where **α0** instructs the format command to insert the ***literal text*** of the 0-th positional value to the right of the format string (and so on for **α*n*** and the ***n***-th positional value); that text is treated as a simple character scalar or vector[8] that is inserted into the format string ***before*** its other components are evaluated. Here we alter the format above to integrate the values directly into the string; while equivalent, this has the advantage of making the format statement itself more concise and easier to read:

```
    ⍝ Student Grades #2: Names, Scores, Hdr, Ftr as above
      ∆f 'α0α1{↑Names} {⍕Scores} {↑Grades}'  Hdr Ftr
[Same output as previous example]
```

Here's another example, where a header (α0) and a footer (α1) both contain embedded Code Fields with formatting specifications:

```
      hdr←'⎕H¨{C19⊂+⊃$"  The⍎◊ Whole ⍎◊ Story"}¨'
      ftr←'⎕F¨{C19⊂¯⊃$""}¨'
      ∆f 'α0α1{F6.3$?2 3⍴0}' hdr ftr
++++++  The  ++++++
++++++ Whole ++++++
++++++ Story ++++++
 0.752 0.462 0.197
 0.912 0.347 0.888
-------------------
```

[8] If an immediate variable **α*n*** cannot be converted to a character string via **⍕α*n***, an error occurs.

Positional vs. Immediate Variables: Positional variables of the form **⍵0** have a different role than immediate variables **⍺0**[9]. Each positional variable is treated as a full-fledged variable useful only in code expressions similarly to ⍵ in APL dfns; it can be of any shape or type that makes sense in the context. In the above example, we could use positional variables for `Names`, `Scores`, and `Grades`. Positional and immediate variables share the same numbering from left to right, based on absolute position to right of the format string (starting at 0). This example uses two immediate and three positional variables:

```
⍝ Student Grades #3:
⍝ This has same value as the Student Grades example above...
∆f '⍺0⍺1{↑⍵2} {⍕⍵3} {↑⍵4}'  Hdr Ftr Names Scores Grades
```

Sequential Positional (⍵⍵) and Immediate (⍺⍺) Variables: As a shortcut, useful in some cases, instead of specifying a format string with positional or immediate variables in sequence (**⍵0 …⍵1...⍵2** or **⍺1...⍺2...⍺3**), **⍵⍵** or **⍺⍵** can be used to refer to the ***next*** positional or immediate variable in sequence, respectively. That is, if **⍵5** was just referenced[10], reading left to right, then a ⍵⍵ to its right will refer to ⍵6; a subsequent **⍵⍵** will refer to **⍵7**.[11] Immediate variables work similarly, based on prior positional variables (each has its own counter, so ⍵⍵ will never depend on prior immediate variables or vice versa).

Thus, cases (a) and (b) below are equivalent:

```
  Name←'John Jones'  ◊ Addr←'41 Maiden Ln'  ◊ Tel←'555-1234'
⍝ Case (a)
  ∆f 'Name {⍵⍵}, Address {⍵⍵}, Tel. Number {⍵⍵}' Name Addr Tel
⍝ Case (b)
  ∆f 'Name {⍵0}, Address {⍵1}, Tel. Number {⍵2}' Name Addr Tel
```

Note that **⍺⍺** is used as an "escape" to enter a single **⍺** into a literal string, but is only required when followed by a digit (i.e. **⍺⍺** alone is simply the literal '⍺⍺') :

```
   ∆f 'Alpha (⍺), Alpha0 (⍺⍺0), Omega (⍵), and Question Mark (⍺⍵)'   '?'
Alpha (⍺), Alpha0 (⍺0), Omega (⍵⍵), and Question Mark (?)
```

Use of standard APL ⍵ (or ⍺) in Code Fields : What if you want to ignore all this business about ⍵0, ⍵⍵, and so on in Code Fields? You can simply use standard APL ⍵ as you normally would. Assuming **⎕IO=0**, (0⊃⍵) will refer to the first item in ⍵, (¯2↑⍵) will gather the last two items, and ⍵ will designate the entire right argument of **∆f**, *after* the format string (which is inaccessible).

```
  ∆f 'The sum of {{{⍵}}} is: {+/⍵} ' 10 20 30 40 ⍝ {{ = literal '{{'
The sum of {10 20 30 40} is: 100
```

---

[9] The highest positional variable is **⍵999**. By default, the highest immediate variable is **⍺9**, to make it easy to directly juxtapose immediate variables and other text, e.g. **⍺99** would map to the string 'K-9' if **⍺9** is 'K-'. The option **⍠A2** will allow immediate variables from ⍺0 to ⍺99. Only **⍠A1** (default) and **⍠A2** are supported.

[10] When ⍵⍵ or ⍺⍵ is used before any other ⍵- or ⍺-reference, it is always ⍵0 or ⍺0, respectively.

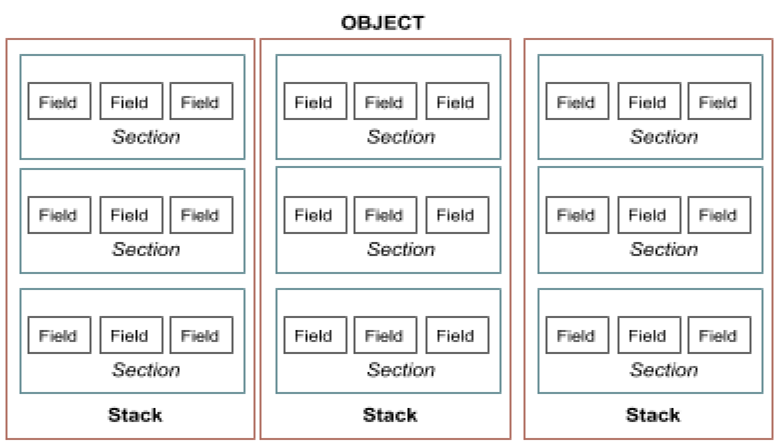
[11] More generally, **⍵⍵** always refers to ***⍵N+1,*** if the prior positional variable was ⍵***N***, whether specified directly (e.g. as ⍵6) or a prior ⍵⍵.

Order of Execution for Code Fields[12] : If there are multiple **Code Fields** within a format string, they are executed in APL order, right to left. Variables set in a **Code Field** (on the right) will be visible further left, but discarded after the format string has been returned. To ensure a variable is visible in the calling environment, either modify it in a **Code Field** (e.g. **a,←' '**) or use global assignment (**∘←**). Here we assign **a** *locally*, then refer to it in the next **Code Field** to the left.

```
      a←10
      ∆f 'The sum of {{{a}}} is: {+/a←⍵} ' 10 20 30 40  ⍝ {{ = literal ‘{{‘
The sum of {10 20 30 40} is: 100
      a    ⍝ “outer” variable not changed.
10
```

Advanced Features:
Fields, Sections, Stacks, and Objects:

Format is designed hierarchically, starting with **fields** (rectangular or character data) built of APL multidimensional and multi-line data and formatting specifications. **Fields** are catenated left-to-right into a 2-D grouping called a **section** that can operate as a unit. In simple cases, there is one **section**. In more complex cases, **sections** are input left to right, then displayed as a **stack,** top to bottom; they can be joined left-justified, centered, or right-justified. A set of **sections**, one on top of the other, constitutes a **stack;** there can be one **stack** in a formatted string, or several. Finally, **stacks** can be aligned together left-to-right, like **fields**, constituting a formatted **object.** An **object** can be decorated with a **header** and a **footer,** centered over the entire object. While no formatted object would sensibly be this complicated, this diagram maps out the hierarchy from field to section to stack to object:



Summary of Object-Building: We’ve shown the building blocks of several kinds of **fields**: **literal fields, code fields,** and **null and space fields. Literal** fields can be built into 2-D blocks using ⍎◊ or ⎕U unicode characters. **Code fields** are structured using formatting specifications (**⎕FMT**-based or extensions) on APL arrays. **Null fields** and **space fields** define horizontal size (in spaces) that automatically match the height (number of lines) of surrounding fields. As those fields are catenated left to right, they produce a **section.**

[12] This applies as well to **Name Fields** and **Quote Fields** (see **Appendix A**).

Sections can be joined together into **stacks**, one over the other using the **section-end delimiter ⍎↓** after each stack (the delimiter may be omitted after the *last* section in the format string).

Finally, **stacks** can be joined left to right into a single **object** (with optional object-level **header** and **footer**) using the **stack-ending delimiter ⍎→** (the delimiter may be omitted after the *last* stack in the format string).

Object-Building Delimiters

Level	Function	Symbols	Builds	Direction
Field	Separator	{}	Section	Horizontal
Section	Delimiter	⍎↓	Stack	Vertical
Stack	Delimiter	⍎→	Object	Horizontal

Advanced Miscellany

Building Sections into Stacks: When building a stack vertically, one section may be wider than another. Options **⍠L, ⍠C,** and **⍠R** are placed within the *upper* section (before the delimiter ⍎↓) to indicate that that section should be placed on the left (**⍠L**), central (**⍠C**), or right (**⍠R**) side of the next section down. By ***default***, sections are stacked ***left-***justified (**⍠L**).

```
      ∆f '⍠R{C11<1>$box 2 3⍴⍳6}⍎↓⍠L{C15<2>$box 2 3⍴⍳6}⍎↓⍠C{C21<3>$box2 3⍴⍳6}'
             1┌ ─ ─ ─ ┐1
             1│ 1 2 3 │1
             1│ 4 5 6 │1
             1└ ─ ─ ─ ┘1
222┌ ─ ─ ─ ┐222
222│ 1 2 3 │222
222│ 4 5 6 │222
222└ ─ ─ ─ ┘222
333333┌ ─ ─ ─ ┐333333
333333│ 1 2 3 │333333
333333│ 4 5 6 │333333
333333└ ─ ─ ─ ┘333333
```

Escaping Formatting Characters: Formatting characters can be escaped (displayed as literals) as follows:

Escape Sequence	Evaluates as	Comments
`{{`	`{`	If not balanced, use **⎕U123.**
`}}`	`}`	If not balanced, use **⎕U125.**
`⍎⍎`	`⍎`	E.g. **⍎⍎◊** displays as **⍎◊.**
⎕⎕U*ddd*	⎕U*ddd*	Bare **⎕U** requires no escaping.
⎕⎕N*ddd*	⎕N*ddd*	Bare **⎕N** requires no escaping.
⍠⍠A	⍠A	Escape required only for ⍠[A-Z].
αα0	α0	**αα** w/o following num is **'αα'.**
⍵0, ⍵1, ...	⍵0, ⍵1, ...	Is literal, outside Code Field.
α*N* **or** α⍵	α*N* **or** α⍵	If *n*-th pos'l arg omitted, α*N* is literal **'α*N*'**, and **α⍵** is literal **'α⍵'.**

Faux Space Option (⍠S*c*): Sometimes it's easier to see what's happening in a literal string with spaces by substituting another, visible character for spaces on input. Likewise, it's sometimes useful to separate out different parts of the format string on input just for clarity without adding extra spacing on output. The **Faux Space** option allows you to address either or both of these goals for the entire format string. **⍠S*c*** specifies that the single character ***c*** will be replaced by a space, wherever it occurs; at the same time, all space characters in the format string are removed (whether in quotes, code, or otherwise). Again, the **Faux Space** applies to the *entire* format string.

```
⍝ Use ? for output space. Note literal spaces are there only for clarity.
     ∆f '⍠S? one?  two?  three  .'
one two three.
   ⍝ Use ? for space, but specify it via an Immediate Variable (‘?’=⎕UCS 63)
     ∆f '⍠Sα0 one?  two?  three .' (⎕UCS 63)
one two three.
```

Appendix A

Setting up run-time access to format strings

To make the format functions and run-time routines available, copy in namespace **format** from file ***format.dyalog*.** You can make format strings available for the current APL session using the following function:

```
  ∇ formatActive
   ⍝ Execute at start of session.
   ⍝ ∘ Enable formatting in the session namespace, ⎕SE.
   :IF 0=⎕SE.⎕NC 'format'
      ⎕PATH,←' ',⍕⎕SE.SALT.Load '-target=⎕SE format'
   :ENDIF
 ∇
```

If you prefer to load **format** permanently into, for example, the top-level namespace #, simply replace (only) the two underscored ***⎕SE*** above with ***#***, or otherwise adjust for your use. Or use the **]Load** command:

```
  ]load format
  )save
```

Loading file **format** will expose functions **∆format** and **∆f**, as well as the background utility function **formatPath,** which will point several other service routines to the active format namespace, minimizing the pollution of the user namespace.

Name Fields and Quote Fields: Similar to Code Fields, f-strings also support two other "experimental" fields, **Name** and **Quote Fields**. They are described only in Appendix A.

```
      NF:   ⍎ [fmt  $$? ] [simple_code | '(' any_code ')' ] name
      QF:   ⍎ [fmt  $$? ] [simple_code | '(' any_code ')' ] quoted_string
```

Where

fmt	a mixture of **fmt1** and **fmt2** specs as for Code Fields[13];
$$?	either $ or $$ as for Code Fields;
simple_code	any APL core functions or operators, integers, and spaces; Names, including system names (**⎕IO**, etc.), are not allowed.
any_code	an arbitrary APL expression within *balanced* parentheses.
name	an APL variable name with optional indexing
quoted_string	a double-quoted or single-quoted string.

Using **simple_code** allows selection, rotating, incrementing, and other simple operations, which can be distinguished from an APL variable expression (name plus index) or quoted string. A typical example of a **Name Field** might be:

[13] Where **fmt1** consists of the standard formatting codes of APL **⎕FMT,** omitting quotes, and **fmt2** consists of special *extensions* **L, R, C** to APL **⎕FMT** for left- and right-justification and centering. The latter may be used with *any* APL extension, but the former must be within **⎕FMT**'s domain.

```
      AR←?2 2 3⍴0
      ∆f '⍛C20$"This is a heading"⍛↓⍛F6.3$  4 3⍴AR'
  This is a heading
  0.092 0.150 0.093
  0.096 0.230 0.567
  0.360 0.228 0.489
  0.561 0.317 0.573
```

Name Fields and **Quote Fields** are *experimental* features, which duplicate the functionality of **Code Fields**. In their most complex syntax, where **(any_code)** is used, some may view it as unwieldy and less elegant or obvious than **Code Fields**, which use braces reminiscent of dfns. On the other hand, in simpler cases, e.g. printing simple values like the following, **Name Fields** and **Quote Fields** may seem very natural to APL users.

```
  ⍝ Example 1
    AR←?2 2 3⍴0
    hdg←'">>> This is a heading<<<"'
    ∆f'⍛C30$⍺0 ⍛↓⎕IO is ⍛⎕IO, AR[1;1;] is ⍛F5.2$$AR[1;1;]' hdg
      >>> This is a heading<<<
⎕IO is 1, AR[1;1;] is  0.48 0.69 0.81

  ⍝ Example 2
    ∆f 'The sum of ⍛⍵ is: {+/⍵} ' 10 20 30 40
The sum of 10 20 30 40 is: 100
```