∆F Strings for Formatting

Description

∆F is a ***prototype*** for a convenient formatting utility for Dyalog APL. Expanding on ideas from other languages, such as Python's *f-strings*, **∆F** displays multiline text, objects of various types, ranks and depth,arbitrary code, and integrated custom formatting, with an ***APL*** flair.
∆F is designed to be useful for assertions (which display values only when boolean conditions are met), debugging, and routine display, expanding on existing capabilities of **⎕FMT** and other APL tools.

Outline

- Getting Started
- ∆F Fields
  - Text Fields are 2D
    - Newlines and other Escape Sequences: `\⋄` and `\{` `\}` `\`
  - Code Fields: Basics
    - Simple Variables
    - Code Fields Are DFNS
      - Performing calculations, etc.
    - Debug Mode
    - DQ Strings
    - Escapes
    - Comments
  - Space Fields
    - Comments
  - Code Fields: Advanced Topics
    - Pseudo-Builtin $ for ⎕FMT formatting
    - Pseudo-builtin $$ for a boxed display
    - ∆F arguments: ⍹0, ⍹, etc.
    - Justification and Centering with $: Beyond ⎕FMT specifications.
    - Self-documenting Code fields: {code → }
    - Code Field namespace ⍺
- Library Routines for Users
- ∆F and Looping
- Assertions with ∆F
- ∆F Syntax

Getting Started

Let's get started!

Let's be sure the file **∆Format.dyalog** is accessible and loaded. When fixed, it creates a single function `∆F`. And let's be sure the environment is set up. ***We'll assume an Index Origin of 0 (`⎕IO←0`) for this session.***

In [1]:
```
⍝ ⎕←  '0. Our active directory is ',⎕SH 'pwd'
  '1. Does ∆Format.dyalog exist in this directory? ', 'No' 'Yes'⊃⍨ ⎕NEXISTS '∆Format.dyalog'
  '2. Loading ∆Format.dyalog as:',(2 ⎕FIX 'file://∆Format.dyalog')
  '3. ','∆F utility now exists!' 'Whoops! No ∆F function.' ⊃⍨3≠⎕NC '∆F'

⍝ Set the usual system vars to something reasonable...
  ⎕IO← 0 ⋄ ⎕ML← 1 ⋄ ⎕PP← 6 ⋄ ⎕FR← 645
```

Out[1]:
```
1. Does ∆Format.dyalog exist in this directory? Yes
```

Out[1]:
```
2. Loading ∆Format.dyalog as: ∆Format
```

Out[1]:
```
3. ∆F utility now exists!
```

Before showing how ∆F works, let's create a couple of variables...
Here are the vectors `string` and `numbers` :

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

We will display them using ∆F.

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

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

We can do this more concisely using **Self-documenting Code Fields** (thanks to Python), signaled by the trailing right arrow *before* the closing curly brace: `{code → }` . Spaces adjacent to the arrow are optional, but may make the formatted output more readable. **Self-documenting Code Fields** are discussed further below. *(Note: The symbol ➤ is a special right arrow that separates the literal code from its value.)*

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

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

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

∆F Fields
Text Fields, Code Fields, and Space Fields

Text Fields: `'A simple string'`

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

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

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

Text Fields are 2D

Like ⎕FMT -formatted objects, text fields are always 2-dimensional (matrices).

In [6]:
```
⍴∆F 'This is a simple string.'
```

Out[6]: 1 24

Text Fields are 2D: 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 2D list of items or a multiline paragraph. ∆F always returns a character matrix of 0 or more rows.

In [7]:
```
∆F 'This\⋄is a\⋄multiline string.'
```

Out[7]:
```
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 `}` . So, if you want to enter a left brace literal, escape it with \, as here.

In [8]:
```
∆F 'This is a diamond ⋄, an opening brace \{, a closing brace \}, and a backslash \\.'
```

Out[8]:
```
This is a diamond ⋄, an opening brace {, a closing brace }, and a backslash \.
```

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 [9]:
```
∆F '"We can sum the numbers ⍵ via +\⍵"'
∆F '"We can sum the numbers ⍵ via +\\⍵"'
```

Out[9]:
```
"We can sum the numbers ⍵ via +\⍵"
```

Out[9]:
```
"We can sum the numbers ⍵ via +\⍵"
```

Code Fields: Basics

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

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

In [10]:
```
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 [11]:
```
∆F '{what} is a {type} {thing}.'
```

Out[11]:
```
This is a simple string.
```

Knowing Your Fields

This ∆F 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' ∆F ...`

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 [12]:
```
'd' ∆F '{what} is a {type} {thing}.'
```

Out[12]:
```
┌────┐┌──────┐┌──────┐┌─┐┌──────┐┌─┐
│This││·is·a·││simple││·││string││.│
└────┘└──────┘└──────┘└─┘└──────┘└─┘
```

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

As shown above, in addition to **Text** fields, we can create executable **Code** fields, using braces {...}. A **Code** field with 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 **∆F** 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 has implicit arguments `⍵` and `⍺` :

- ⍵ consists of all the arguments to the full **∆F**. These can be accessed individually via ⍹0, ⍹1, etc (see below).
- ⍺ is a reference to a special namespace with a small library and a place to share temporary variables across **Code** fields (see below).

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

Code Field Comments: `{... ⍝...}`

A Code field may include one or more comments. Each comment starts with a lamp `⍝` and contain no braces or ⋄; it terminates just before a following diamond `⋄` or a closing brace `}` .

For example:

In [13]:
```
∆F '{ ?0  ⍝ A random number ⍵: 0<⍵<1 }'
```

Out[13]:
```
0.193673
```

Code Fields Are DFNS: More Complex Examples

Let's look at more complex examples. First, what if a variable itself is more than a simple one-line text string? Unlike format strings in other languages, **ΔF strings** handle multi-dimensional objects the *APL way*!

In [14]:
```
nums←  ⍪⍳3
what←  ↑'This' 'That' 'The other thing'
type←  ↑'simple' 'hard' 'confusing'
thing← ↑'string' 'matrix' 'thingamabob'
ΔF '{nums} {what} is a {type} {thing}'
```

Out[14]:
```
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 the three variables above. 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. (We'll say more about **DQ Strings** in a moment.)

In [15]:
```
what←'This' 'That' 'The other thing'
type←'simple' 'hard' 'confusing'
thing←'string' 'matrix' 'thingamabob'
ΔF '{ ⍪⍳≢what }  { ↑what } is a { ↑type } { ↑thing,¨"." }'
```

Out[15]:
```
0  This            is a simple    string.
1  That                 hard      matrix.
2  The other thing      confusing thingamabob.
```

Code Fields Support Arbitrary Calculations

As you have already seen, ΔF **Code** fields can be used for arbitrary calculations:

In [16]:
```
r←1 2 5     ⍝ radius
script←'If the radius is { ⍪r }m, the circumference is { ⍪○2×r }m, and the area is { ⍪○r*2
ΔF script
```

Out[16]:
```
If the radius is 1m, the circumference is  6.28319m, and the area is  3.14159m².
                 2                        12.5664                     12.5664
                 5                        31.4159                     78.5398
```

As indicated above, **Code** fields are executed sequentially from left to right as independent dfns:

In [17]:
```
ctr←1
ΔF 'ctr is {ctr}, now {ctr⊣ctr+←1 }, now {ctr⊣ctr+←1 }, now {ctr⊣ctr+←1 }'
```

Out[17]:
```
ctr is 1, now 2, now 3, now 4
```

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

Quotes'

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 in ∆F **strings**. Single quotes *may* appear, most usually as literal characters, rather than to create strings.

DQ Strings: Escapes, including `\⋄` ...

DQ Strings support the escaped sequences `\⋄` and `\\`. `\⋄` is a convenient way to enter newlines (actually ⎕UCS 13, the carriage return character) into linear strings (APL character vectors). When the ∆F *string* is printed, newlines will create separate *rows* in the output matrix.
(Note: Unlike **Text** fields, **DQ Strings** in **Code** fields do not require or allow braces to be escaped: braces are ordinary characters inside **DQ Strings**.)

To include an actual double quote within a **DQ String**, double the doublequote (""), just as one would for *single* quotes in standard APL strings. (As always, APL of course requires single quotes within character strings to be doubled on entry). Notice how the string below is a 3-row matrix, one row for each line of the **DQ String**.

In [18]:
```
⍝                      Row 1   Row 2                   Row 3...
  '# rows:', ≢⎕← ∆F '{"This \⋄is a ""DQ"" example,\⋄isn''t it?"}'
```

Out[18]:
```
This
is a "DQ" example,
isn't it?
# rows: 3
```

Space Fields: `{   }`

The third and last field type is a **Space** field, which looks just like an *empty* **Code** Field, containing only zero or more spaces between the braces `{ }` and, optionally, a comment. A **Space** field forms a distinct field and is a good way to separate other sorts of fields, i.e. **Text** or **Code** fields.

Space Field Comments: `{  ⍝...}`

A **Space** field may include a comment, which starts with a lamp `⍝` and contains ***no*** braces; the comment ends just before the right brace `}` that terminates the **Space** field.

For example:

In [19]:
```
⍝ Show Space Field comments...
    ∆F '{⍪1 2 3}{     ⍝ 5 spaces }{⍪○1 2 3}'
'd' ∆F '{⍪1 2 3}{     ⍝ 5 spaces }{⍪○1 2 3}'
```

Out[19]:
```
1      3.14159
2      6.28319
3      9.42478
```

Out[19]:
```
┌─┐┌─────┐┌───────┐
│1││·····││3.14159│
│2│└─────┘│6.28319│
```

```
|3|        |9.42478|
└─┘        └───────┘
```

Space Field Examples

In [20]:

```
⍝ ...
    ∆F 'This is{ }a test.'
'd' ∆F 'This is{ }a test.'
```

Out[20]:

```
This is a test.
```

Out[20]:

```
┌───────┐┌─┐┌───────┐
│This·is││·││a·test.│
└───────┘└─┘└───────┘
```

But why bother with space fields at all?

- They are useful when separating one multiline strings or code field from the next; even a zero-width space field can separate two **Text** fields;
- 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 [21]:

```
⍝ ...
    ∆F 'This\⋄is a\⋄multiline\⋄field!{ ⍝ 1 Space}{"This\⋄is\⋄as well!"}'
'd' ∆F 'This\⋄is a\⋄multiline\⋄field!{ ⍝ 1 Space}{"This\⋄is\⋄as well!"}'
```

Out[21]:

```
This      This
is a      is
multiline as well!
field!
```

Out[21]:

```
┌─────────┐┌─┐┌────────┐
│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 [22]:

```
∆F '1. \⋄2.\⋄3.{}Jane\⋄John\⋄Nancy'
```

Out[22]:

```
1. Jane
2. John
3. Nancy
```

This is equivalent, with an explicit Space field of length 1.

```
∆F '1.\⋄2.\⋄3.{ }Jane\⋄John\⋄Nancy'
```

Space Fields with explicit width counts: `{  :30: }`

Sometimes it's a bit inconvenient to count out a field width or to specify a wide **Space Field**. If so, rather than typing the actual spaces, simply enter inside the braces the desired number of spaces ***as an integer*** between two colons (possibly followed by a comment). The colons are there to avoid confusion with **dfn** syntax. Each explicit width count must between ***0 and 999***.

Here we separate each **Code Field** by exactly five spaces, encoded *slightly* differently:

In [23]:
```
⍝ ...            12345        Five          Still Five Spaces
    ∆F '{⍪1 2}     {⍪○1 2}{ :5: }{⍪*1 2}{:5:⍝Five spaces}{⍪(○1 2)××1 2}'
'd' ∆F '{⍪1 2}     {⍪○1 2}{ :5: }{⍪*1 2}{:5:⍝Five spaces}{⍪(○1 2)××1 2}'
```

Out[23]:
```
1     3.14159     2.71828      8.53973
2     6.28319     7.38906     46.4268
```

Out[23]:
```
┌┐ ┌─────┐ ┌───────┐ ┌─────┐ ┌───────┐ ┌─────┐ ┌───────┐
│1││·····││3.14159││·····││2.71828││·····││·8.53973│
│2│└─────┘│6.28319│└─────┘│7.38906│└─────┘│46.4268·│
└┘        └───────┘        └───────┘        └───────┘
```

(More on) Code Fields: Advanced Topics

Pseudo-builtin $ for Detailed Formatting

Dyalog APL has a built-in formatting utility ⎕FMT , which may be old-fashioned but is rather essential for precise formatting of numerical data (and sometimes text data). To make ⎕FMT and its standard parameters easier to use, we provide it as the pseudo-builtin $, with extensions described below. Here's a typical example, where the (optional) formatted left argument (⍺) is placed within a DQ string (as required with ∆F **Code** Fields):

In [24]:
```
∆F '{ "F8.6" $ ?3⍴0  ⍝ Random Floats }  {"ZI2,⊂/⊃,ZI2,⊂/⊃,ZI4" $ ⌽⍪⎕TS[2 1 0] ⍝ a European
```

Out[24]:
```
0.921368  13/12/2021
0.219363
0.936951
```

Keep in mind that $, like ⎕FMT , treats simple vectors as column vectors (1-column matrices) ***by default***. (**∆F** has a way to override this default: see below.)

Let's move on to another example with ⎕FMT features you may have forgotten.

In [25]:
```
∆F 'Multiples of pi: {"I1,⊂×π =⊃" $ 1+⍳4} {"F10.7,⊂…⊃" $ ○1 2 3 4}'
```

Out[25]:
```
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 [26]:
```
'd' ∆F 'Multiples of pi: {"I1,⊂×π =⊃" $ 1+⍳4} {"F10.7" $ ○1 2 3 4}'
```

Out[26]:
```
┌─────────────────┐┌─────┐┌┐┌──────────┐
│Multiples·of·pi:·││1×π·=││·││·3.1415927│
```

```
└───────────────────┘ │2×π·=│ └─┘ │·6.2831853│
                      │3×π·=│     │·9.4247780│
                      │4×π·=│     │12.5663706│
                      └─────┘     └──────────┘
```

Pseudo-Function $$ for a Boxed Display

If we want a **Code** field to be boxed in the regular output, we can use the pseudo-builtin function **$$**. By default (no left argument or a left-argument of 0), no middle dots (·) appear with $$. If you want middle dots to appear in place of spaces, you must provide a left argument of 1.

In [27]:
```
OVER← (⍪,⍥⊂)                                         ⍝ A Helper Function
boats← (↑'Nina' 'Pinta' 'Santa Maria')

title← ∆F 'Default (Spaces) { :7: } With Middle Dots'
bfmt←  ∆F ' { $$ ⍵1} {:10:} { 1 $$ ⍵1 }' boats      ⍝ Hold on! We explain ⍵1 just below.
title OVER bfmt
```

Out[27]:
```
 Default (Spaces)          With Middle Dots

  ┌───────────┐              ┌───────────┐
  │Nina       │              │Nina·······│
  │Pinta      │              │Pinta······│
  │Santa Maria│              │Santa·Maria│
  └───────────┘              └───────────┘
```

∆F Code field arguments: `⍹0` ... `⍹99` and `⍹` (or `⍵0` ... `⍵99` and `⍵_`)

∆F allows **Code** fields to access elements in its right argument, including the format string itself. Elements here refer to **top level scalar objects** in the right argument `⍵` to **∆F**, normalized to a nested form `⊆⍵`, ordered scalar by scalar (in ⎕IO=0) as (0⊃⍵) (1⊃⍵) (2⊃⍵) ... (N⊃⍵):

- The format string itself `(0⊃⍵)` is abbreviated as `⍹0`, `(1⊃⍵)` as `⍹1`, ..., (N⊃⍵) as `⍹N`.
- If a Code field refers to an element that does not exist, a runtime `INDEX ERROR` signal is generated as expected:
```
     ∆F '{⍹2}' 1
 ∆F INDEX ERROR: ⍹2 is out of range.
```
- The character `⍹` (omega underscore) is `⎕UCS 9081`.
- If `⍹` is not handy, there are alternatives using a simple omega `⍵`:
  - for omega underscore with numeric suffixes, use:
    - `⍵0` for `⍹0`, `⍵10` for `⍹10`, etc.
  - for bare omega underscore (next element-- see below), use:
    - `⍵_` for `⍹`. That's a ⍵ followed by an underscore _.

Here is an example accessing `⍹1`, shorthand for `(1⊃⍵)`.

In [28]:
```
∆F '{⍹1} multiples of pi: {"I1,⊂×π =⊃" $ 1+⍳⍹1 } {"F10.7" $ ○ 1+⍳⍹1}'  3
∆F '{⍹1} multiples of pi: {"I1,⊂×π =⊃" $ 1+⍳⍹1 } {"F10.7" $ ○ 1+⍳⍹1}'  2
```

Out[28]:
```
3 multiples of pi: 1×π =   3.1415927
                   2×π =   6.2831853
                   3×π =   9.4247780
```

Out[28]:
```
2 multiples of pi: 1×π =   3.1415927
```

```
                        2×π =   6.2831853
```

The symbol ⍵ used alone in a **Code** field will select the *next* argument in sequence:

- ⍵ means ⍵1 if this is the first use in any Code field of either an explicit ⍵N or bare ⍵, else
- ⍵ means ⍵N+1 if ⍵N was the most recently accessed:
  - For example, ⍵ references ⍵5 if ⍵4 was the most recently accessed, explicitly or implicitly.

This makes it easy to format a set of items:

In [29]:

```
w1F← 'F4.2,⊂%⊃'
w2D← 2.200 3.834 5.996
w3F← '⊂£⊃,F7.2'
w4D← 1000.23, 2250.19 2500.868
⍝            ⍵1   ⍵2                  ⍵3   ⍵4    ⍵1    ⍵2    ⍵3     ⍵4
∆F 'Rate: {⍵ $ ⍵}; cur. value: {⍵ $ ⍵}'  w1F   w2D   w3F    w4D
```

Out[29]:

```
Rate: 2.20%; cur. value: £1000.23
      3.83%              £2250.19
      6.00%              £2500.87
```

Note also that ⍵0 can never be selected via the lone ⍵ (as the *next* ⍵ argument), since the *last* index specified is never less than 0, so the *next* is never less than 1. In short, if you want ⍵0 , you type it explicitly!

Pseudo-Function $: Justification and Centering Codes *L*, *C*, *R*
Vector Arg treated as Column Vector

The pseudo-function $ has been extended with special codes for justifying or centering objects within a **Code** field. (Codes *l*, *c*, *r*— are discussed further below.)

	Codes *L, C, R*	
Justif. Type	**Col Vec Preferred**	**Row Vec Preferred**
left	**L***nnn*	**l***nnn*
center	**C***nnn*	**c***nnn*
right	**R***nnn*	**r***nnn*

The digits nnn , a 1- to 3-digit number, indicates the minimum width of the field. If a field is already wider than nnn characters, the field is left as is. These codes are valid with either *numeric* or *text* arguments. 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 conventions of dyadic ⎕FMT).

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

In [30]:

```
names←↑'John' 'Mary'
      ∆F '<{"L10" $ ⍵1}> <{"C10" $ ⍵1}> <{"R10" $ ⍵1}>' names
'd'   ∆F '<{"L10" $ ⍵1}> <{"C10" $ ⍵1}> <{"R10" $ ⍵1}>' names
```

Out[30]:

```
<John      > <   John   > <      John>
 Mary            Mary            Mary
```

Out[30]:

```
┌─┐┌──────────┐┌───┐┌──────────┐┌───┐┌──────────┐┌─┐
│<││John······││>·<││···John···││>·<││······John││>│
```

```
└─┘ │Mary······│ └───┘ │···Mary···│ └───┘ │······Mary│ └─┘
    └──────────┘       └──────────┘       └──────────┘
```

Here, we format a couple of numeric fields, one centered automatically via $ code `C25` and the other manually via a *standard* **⎕FMT** code `X6`, which adds explicit spacing to build the same field width; both do the job:

In [31]:

```
title← ∆F '{     }Use ∆F Extension C25\⋄   25 chars wide{ :13: }Use ⎕FMT X6: 6+13+6=25\⋄
centr← ∆F '{1$$ "C25,ZF13.9" $ ⍵1 } versus {1$$ "X6,ZF13.9,X6" $ ⍵1 }' (○2 20 300)
title OVER centr
```

Out[31]:

```
     Use ∆F Extension C25                Use ⎕FMT X6: 6+13+6=25
         25 chars wide                        25 chars wide
┌─────────────────────────┐ versus ┌─────────────────────────┐
│······006.283185307······│        │······006.283185307······│
│······062.831853072······│        │······062.831853072······│
│······942.477796077······│        │······942.477796077······│
└─────────────────────────┘        └─────────────────────────┘
```

Note also the use of the explicit digits in the **Space** field `{   :13:   }` to insert a significant set of spaces to separate the titles.

Pseudo-function `$` : Justification and Centering Codes *l*, *c*, *r*
Vector arg treated as *Row* 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 (`nnn` is a 1- to 3-digit number):

Codes *l*, *c*, *r*

Justif. Type	Col Vec Preferred	Row Vec Preferred
left	**L***nnn*	**l***nnn*
center	**C***nnn*	**c***nnn*
right	**R***nnn*	**r***nnn*

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 codes `"c0"`, `"C0"`, *et cetera* do the job because justification and centering codes ensure a **minimum** width, never truncating fields over that width.

In [32]:

```
∆F 'For n∊1 2 3, nπ = { "c0" $ ○ ⍵1 }.   {"I1,⊂π = ⊃" $ ⍵1} { "C0" $ ○⍵1 }' (1 2 3)
```

Out[32]:

```
For n∊1 2 3, nπ = 3.14159 6.28319 9.42478.   1π =   3.14159
                                             2π =   6.28319
```

```
                                3π  =  9.42478
```

Code Fields: Self-Documenting Code Fields `{ Code → }`

As mentioned earlier, a **Code** Field can be made **self-documenting** by inserting a right arrow → just before the closing right brace. In more detail, the entire code of the **Code** Field, including the right arrow *and* the spaces *before* and *after* it, will be included in the formatted output, followed by the executed code. For clarity,, the APL right arrow → is replaced by a special right arrow ➤ .

In [33]:
```
MyString← ↑'This' 'is my' 'string'
Today←  ⍪⎕TS
∆F '<{MyString → }>  <{ 3↑Today → }>.  <{"I4" $ 3↑Today  →  }>'
```

Out[33]:
```
<MyString ➤ This   >  < 3↑Today ➤ 2021>.  <"I4" $ 3↑Today  ➤  2021>
            is my               12                             12
            string              13                             13
```

Warning: **Jupyter Notebook** seems to improperly format some APL output, including output from self-documenting **Code** fields. This does not occur in a Dyalog APL **Ride** session.

Comments in Self-Documenting Code Fields

Comments are allowed in **Self-documenting Code Fields**, but must be terminated by a diamond ⋄ ; the final right arrow appears just before the right brace that closes the **Code** Field. For example:

In [34]:
```
∆F ' { ⎕AV ⍳"aeiou" ⍝ Find the "vowels" in ⎕AV ⋄  → }  '
```

Out[34]:
```
 ⎕AV ⍳"aeiou" ⍝ Find the "vowels" in ⎕AV ⋄  ➤ 17 21 25 31 37
```

Code Fields: Left argument ⍺ contains special namespace

∆F passes a reference to a special namespace as the left argument ⍺ to each **Code** field. That namespace contains support functions and variables for **∆F**. Names *beginning* with any of the following letters

- an underscore, ⍙, or lower-case letters (*a-z plus*),
- *i.e.* any of the following: _ ⍙ a b c d e f g h i j k l m n o p q r s t u v w x y z þ ã ì ð ò õ à á â ä å æ ç è é ê ë í î ï ñ ó ô ö ø ù ú û ü

are reserved for the application user (you) to use in temporary functions, operators, or variables. One potential use is setting state that is maintained across all **Code** fields (left to right) during the execution of **∆F**, without cluttering the calling environment:

In [35]:
```
⎕FR ⎕PP← 1287 34
    PiSilly ←{ ⍺._PI← ○ ⋄ ⍵ }
    ∆F '{ ⍺ PiSilly 10 }{ ⍺._PI 1 }'
⎕FR ⎕PP←645 10
```

Out[35]:
```
3.141592653589793238462643383279503
```

∆F and Looping

While languages like Python might tend to examples requiring explicit loops like this:

```
 # Python ...
table = [4127, 4098, 7678]
for num in table2:
    print(f'{num:10}')
```

APL can typically format an entire object in a single statement and in a readable format:

In [36]:
```
table← 4127 4096 7678                          ⍝ Could be much bigger, of course...
∆F '{"[]" ⍺.QT ⍪1+⍳≢⍵1}  {⍪⍵1}' table          ⍝ ⍺.QT defined below...
```

Out[36]:
```
[1]  4127
[2]  4096
[3]  7678
```

Or, compare this Python example:

```
 # Python
print(f'Number Square Cube')
for x in range(1, 11):
     print(f'{x:2d} {x*x:3d} {x*x*x:4d}')
```

and its APL equivalent below. *(Of course, a simple Dyalog ⎕FMT statement would work perfectly and concisely in this case!)*

In [37]:
```
rangeV ← 1+⍳10
Head←'Number     Square     Cube'
Head OVER ∆F  '  {"I2" $ ⍵1}       {"I3" $ ⍵1*2}       {"I4" $ ⍵1*3}'  rangeV
```

Out[37]:
```
 Number     Square     Cube
     1          1         1
     2          4         8
     3          9        27
     4         16        64
     5         25       125
     6         36       216
     7         49       343
     8         64       512
     9         81       729
    10        100      1000
```

Keep n mind as you read examples from Python or Javascript that APL is likely to have more intuitive solutions that do not require explicit looping with **∆F**.

∆F for assertions

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

- If the assertion contains *no numeric zeroes*, it is considered **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 considered **false**. It does nothing, quickly returning a shy *0*.

In [38]:
```
⍝ Here, (var<100) is FALSE, so no ∆F string message is produced. 0 is returned.
  var←100
  rc←(var<100) ∆F 'Warning! Variable "var" is out of expected range: var={var}'
  'Assertion formatted (and printed) nothing and returned SHY',rc
```

Out[38]:
```
Assertion formatted (and printed) nothing and returned SHY 0
```

In [39]:
```
⍝ Now, the assertion (var<100) is TRUE, so a ∆F string message is printed. 1 is returned.
⍝ We'll show it explicitly, even though it is shy.
  var←50
  rc←(var<100) ∆F 'Warning! Variable "var" is out of expected range: var={var}'
  '∆F formatted and printed text (via ⎕←) and returned SHY',rc
```

Out[39]:
```
Warning! Variable "var" is out of expected range: var=50
```

Out[39]:
```
∆F formatted and printed text (via ⎕←) and returned SHY 1
```

Library Routines for Users

In addition to `$` and `$$`, **∆F** provides access to several library routines for use in *Code* fields. A *reference* to the library namespace is passsed in `⍺`. The routines are:

```
      [opts] ⍺.FMTX obj
         An extended ⎕FMT. Equivalent to $. See $ for details.
      ⍺.BOX obj
          Display obj in char form using dfns.box. Equivalent to $$. See $$ for
    details.
      [opts] ⍺.BBOX obj
          Display obj in char form using dfns.box, replacing blanks with a
    middle dot (·).
          See also the 'DEBUG' option.
          opts: If opts is specified, it must be a character scalar, which will
          replace blanks in the displayed right argument ⍵.
      [opts] ⍺.QT obj
          Place quotes as defined in <opts> around each line of <obj>
          (via ⎕FMT, if not already a matrix).
          - Quotes are placed as close as possible to non-blanks on left and
    right of
          - each line of the object, extending it left and/or right to add the
            quotes if required.
          obj:  A string scalar, vector, matrix, or vector of vectors.
          opts: opts includes either 1 or two characters or numbers.
          - If one, it is used for both.
          - If a number, it is the Unicode representation for the required
    character.
          - If opts is omitted, a double quote (") is assumed (⍺←'"').
```

UNDER EVALUATION

```
      ⍺.TITLE string
           Displays string with each "word" in Title Case, i.e. with each
           word's first letter capitalized.
```

In [40]:

```
guillemets← '«»'                    ⍝ guillemets: French-style quotes!
gUnicode←   171 187                 ⍝ Unicode for «»
∆F '{⍺.QT ⍵1 ⍝ Default qts}     {guillemets ⍺.QT ⍵1}     {gUnicode ⍺.QT $ ⍵1}' (⍪100×⍳3)
```

Out[40]:

```
  "0"       «0»      «0»
"100"     «100»    «100»
"200"     «200»    «200»
```

In [41]:

```
animals←↑'cats' '  rats'  'dogs   '          ⍝ Note arbitrary blanks within each row of anima
∆F '{animals  } {:10:} {  ⍺.QT animals  } {:10:} { """",animals,""""}'
```

Out[41]:

```
cats                "cats"              "cats   "
  rats                "rats"            "  rats"
dogs                "dogs"              "dogs   "
```

Here's a clever use of `⍺.QT` .

Unlike `"I2"` (a **⎕FMT** specification), `⍺.QT` automatically adjusts to the width of the numbers being formatted. (**⎕FMT** will insert asterisks (***), if the field width is insufficient.

In [42]:

```
∆F '{ " ." ⍺.QT ⍪⍵1}  { "I2,⊂.⊃" $ ⍵1}  { "I3,⊂.⊃" $ ⍵1}   ' (98 99 100)
```

Out[42]:

```
 98.  98.   98.
 99.  99.   99.
100.  **.  100.
```

∆F Syntax

Syntax: **result ← [[assertion | options]] ∆F format_string [arg1 [arg2 ...]]**

- **assertion** ⍺: a homogeneous numeric array.
  - The assertion is "TRUE", unless it contains at least one 0.
    - If TRUE, **∆F** prints the formatted result and returns a shy 1.
    - If FALSE, **∆F** does nothing, returning a shy 0.
- **options** ⍺: DEBUG | COMPILE | HELP | DEFAULT*
  - DEBUG: Displays each field separately (via dfns "box") 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.
    - See **COMPILE option** below.
  - HELP: Displays HELP documentation (⍵ ignored): `∆F⍨'HELP'`
  - DEFAULT: Returns a formatted 2-D array according to the format_string specified.
    - If ⍺ is omitted or ⍺ is a null string `''` , option DEFAULT is assumed.
  - Options may be in either case and abbreviated.
- **format_string** `⍵0` (0⊃⍵)
  - Contains the simple format "fields" that include strings (**Text** fields), code (**Code** fields), and spacing (**Space** fields).
    Code fields support a shorthand for manipulating objects using **$**, **$$**, or **$$$**:

- $ does numeric formatting (via **⎕FMT**), along with justification and centering;
- **$$** displays data using dfns **DISPLAY**;
- **$$$** adds quotes or other decorators around user data (see **QT** above).

- arbitrary **args** `1↓⍵`
  - Optional arguments to be referenced in Code Fields.
  - Shorthand in Code Fields
    - **⍵0** refers to `0⊃⍵` , the ∆F format string.
    - **⍵1** refers to `1⊃⍵` , the first "user" element.
    - **⍵N** refers to `N⊃⍵` , the Nth "user" element (where N is a 1- or 2-digit number)
    - **⍵** by itself refers to the NEXT element, counting from left to right across all code fields.
  - Allowed synonym for **⍵**: You may use `⍵0` for **⍵0**, `⍵N` for **⍵N**, and `⍵_` for a **⍵** by itself.

- **result**
  - For an assertion, result is either a shy 1 (TRUE) or 0 (FALSE, where *at least* one element of ⍺ was 0).
  - The final formatted object is printed (via ⎕←) if the result is TRUE.
  - Otherwise, if `DEFAULT` formatting is specified (*i.e.* the option `COMPILE` is not used), result will be a matrix containing all the fields glued together.
  - If the `COMPILE` option is specified, result will be a code string that can be executed to produce the result either of an assertion (see above) or the `DEFAULT` formatting. It may be executed immediately via ⍎ or converted to a fn (via ⍎) once and then called, e.g. in an implicit or explicit loop.

```
myFormat←  ⍎'COMPILE' ∆F '...Make use of variable myVar ...'
:FOR myVar :in CreateVars 1E4
   myFormat '' elem1 elem2 ...
:ENDFOR
```

- **COMPILE option**: Given a dfn generated via a call to **∆F** via the **COMPILE** option...
  - The first (or only) element of its **right** argument `⍵` may be
    - a nullstring `''`
      - *i.e.* a *dummy* format string which will be ignored; the *original* format string presented at compile time will be available as `⍵0`
    - an alternate format string specified at execution time
      - which will be available instead as `⍵0` .
    - In general: myFun← ⍎ 'compile' ∆F 'my code' (myFun '' [⍵1 ...]) ≡ ('default' ∆F 'my code' [⍵1 ...])
  - Its **left** argument `⍺` is omitted, unless the formatting dfn is an assertion.
    - If ⍺ is present, it must be an **assertion** (see above) with simple numeric data.

> Note: **∆F** is a ***prototype***, depending heavily on regular expressions (via **⎕R** and **⎕S**, including some rather recursive patterns) so it's rather ***slow***. **Compiled ∆F** strings (namely, associated ***dfns*** generated via the *compile* option) are several-fold faster than **standard ∆F** strings; they may be useful in loops or oft-repeated expressions. Nothing substitutes, however, for moving from a ***prototype*** version to ***production*** code.