QUICK IDL TUTORIAL NUMBER TWO: IDL DATATYPES AND ORGANIZATIONAL STRUCTURES
March 3, 2009

Contents

By the term *datatype* we mean, for example, integers, floating point variables, strings, complex numbers. By the term *organizational structures* we mean scalars, vectors, arrays, and structures. We cover these in the following sections. All available datatypes can be arranged in all available organizational structures. For example, we can have arrays of strings, vectors of complex numbers.

1. DATATYPES

Here we cover only the basic IDL datatypes. There are others, including unsigned integers and complex numbers.

1.1. DIGITS, BITS, BYTES, AND WORDS

We have gotten to the place where you need to know a little about the internal workings of computers. Specifically, how the computer stores numbers and characters.

Humans think of numbers expressed in powers-of-ten, or *decimal numbers*. This means that there are 10 digits ($0 \rightarrow 9$) and you begin counting with these digits. When you reach the highest number expressible by a single digit, you use two digits and generate the next series of numbers, $10 \rightarrow 99$. Let f and s be the *first* (1) and *second* (0) digits, respectively; then the number is $f * 10^1 + s * 10^0$. And so on with more digits.

Fundamentally, all computer information is stored in the form of *binary numbers*, meaning powers-of-two. How many digits? Two! They are 0 and 1. The highest number expressible by a single digit is 1. The two-digit numbers range from 10 to 11; the number is $f * 2^1 + s * 2^0$. And so on with more digits. But wait a minute! The word "digit" is a misnomer—it implies something about 10 fingers. Here it's the word **bit** that counts. Each binary "digit" is really a **bit**. So the binary number 1001 is a 4-bit number. What decimal number does the binary number 1001 equal?

For convenience, computers and their programmers group the bits into groups of eight. Each group of 8 bits is called a **byte**. Consider, then, the binary number 11111111; it's the maximum-sized number that can be stored in a byte. What is this number?

Finally, computers group the bytes into **words**. The oldest PC's dealt with 8-bit words—one byte. Many present-day PCs use 32-bit words—four bytes. Macs and the more capable PCs deal with 64-bit words—8 bytes. What's the largest number you can store in a 4-byte word? And how about negative numbers?

Below we describe how IDL (and everybody else) gets around this apparent upper limit on numbers. They do this by defining different data types. Up to now, the details didn't matter much. But now... We don't cover all datatypes below—specifically, we omit Complex (yes, complex numbers!), Hexadecimal, Octal; you can look these up if you are interested.

1.2. INTEGER DATATYPES IN IDL

Integer datatypes store the numbers just like you'd expect. IDL supports integers of four different lengths: 1, 2, 4, and 8 bytes. The shorter the word, the less memory required; the longer the word, the larger the numbers can be. Different requirements require different compromises.

1.2.1. 1 byte: The Byte Datatype

The Byte datatype is a single byte long and always positive. Therefore, its values run $0 \rightarrow 255$. The byte datatype isn't all that useful for general computation, but it serves two essential purposes inside your computer:

1. Images are always represented in bytes. The *data* might not be in bytes, but the numbers that the computer sends to the video processor card are always bytes. Video screens require lots of memory and really quick processing speed, so bytes are ideal. When you write an image to the screen, which is done with the `tv` command in IDL, you send a 2-d array—and those numbers must be bytes. Usually your image is calculated as floats (ordinary numbers with decimal points; see §1.3 below); convert these to bytes using IDL's `bytscl` command.

2. Characters (like the string variable 'a') are represented as bytes. This means that computers only have 255 chacters that can be used in string variables (see §1.6 below).

You can generate a byte array using **bindgen**; you can generate a single byte variable by saying **x=3b**. If, during a calculation, a byte number exceeds 255, then it will "wrap around"; for example, 256 wraps to 0, 257 to 1, etc.

1.2.2. 2 bytes: Integers and Unsigned Integers

With 2 bytes, numbers that are always positive are called **Unsigned Integers**. They can range from $0 \rightarrow 256^2 - 1$, or $0 \rightarrow 65535$. You generate an array using **uindgen**. How do you think unsigned integers wrap around?

Normally you want the possibility of negative numbers and you use **Integers**. The total number of integer values is $256^2 = 65336$ $(= 2 \cdot 32768)$. One possible value is, of course, zero. So the number of negative and positive values differ by one. The choice is to favor negative numbers, so Integers cover the range $-32768 \rightarrow 32767$. You generate an array using **indgen**. What happens with wraparound? What if **x=5**, **y=30000** and **z=x*y**? Check it out!

1.2.3. 4 bytes: Long Integers and Unsigned Long Integers

The discussion here is exactly like that for 2-byte integers, except that 256^2 becomes 256^4. What are the limits on these numbers? See IDL help under "Data Types" and "Integer Constants" for more information. You generate arrays using **ulindgen** and **lindgen**.

1.2.4. 8 bytes: 64-bit Long Integers and Unsigned 64-bit Long Integers

The discussion here is exactly like that for 2-byte integers, except that 256^2 becomes 256^8. What are the limits on these numbers? See IDL help under "Data Types" and "Integer Constants" for more information. You generate arrays using **ul64indgen** and **l64indgen**.

1.3. FLOATING DATATYPES IN IDL

The problem with integer datatypes is that you can't represent anything other than integral numbers—no fractions! Moreover, if you divide two integer numbers and the result should fractional, but it won't be; instead, it will be rounded down (e.g. $\frac{5}{3}$ is calculated as 1). To get around this, the *floating* datatype uses some of the bits to store an *exponent*, which may be positive or negative. You throw away some of the precision of the integer representation in favor of being able to represent a much wider range of numbers.

1.3.1. 4 bytes: Floats

"Floating point" means floating decimal point—it can wash all around. With Floats, the exponent can range from about $-38 \rightarrow +38$ and there is about 6 digits of precision. You generate an array using **findgen** and a single variable by including a decimal point (**x=3.**) or using exponential notation (**x=3e5**).

Printing floating point numbers to the full native precision, instead of what IDL regards as "convenient", requires using a `format` statement in the `print` (or, when annotating a plot, the `xyouts`) command. For example:

```
a= 1.23456789
print, a, format='(f20.10)'
```

prints out 10 decimal points and 20 characters including the decimal point. Of course, we've defined `a` to higher precision than single-precision float carries, so the last bunch of numbers beyond the decimal won't be correct. To make that happen, you need...

1.3.2. 8 bytes: Double-Precision

Like Float, but the exponent can range from about $-307 \rightarrow +307$ and there is about 16 digits of precision. You generate an array using **dindgen** and a single variable by writing **x=3d** or **x=3d5**. Then, when you do the following, it works:

```
a= 1.23456789d0
print, a, format='(f20.10)'
xyouts, xloc, yloc, string( a, format='(f20.10)')
```

For annotating a plot with `xyouts`, you explicitly convert the number to a string of the specified format; the `print` statement does this too, but it's transparent because it automatically does it for you.

1.4. COMPLEX NUMBERS

A complex number C consists of a pair of real numbers, one the real and one the imaginary part [in IDL: `real_part(C)` and `imaginary(C)`]. The two numbers are always floats and can be either single- or double-precision. You generate a single-precision complex number $(2.2 + \imath 3.9)$ with: `C = complex( 2.2, 3.9)` and a double-precision one with `dcomplex`. The arguments can be arrays, giving you a complex-number array. There are also the complex analogues `cindgen` and `dcindgen` (like `findgen` and `dindgen`); and `complexarr` and `dcomplexarr` (like `fltarr` and `dblarr`).

1.5. ILLEGAL NUMBERS

As we've seen, there are limits to the size of digitally-stored numbers. IDL follows the IEEE standard when calculations produce out-of-bound numbers:

1. A floating-point number that exceeds the upper limit is printed out as the string `Inf` (but it is stored as a float).

2. A float that is smaller than the lower limit is returned as zero and the statement "Program caused arithmetic error: Floating underflow" is written on the screen (unless the error reporting is suppressed).

3. An illegal number, such as $(-2)^{1/2}$, is printed out as the string `NaN` (but it is stored as a float).

4. An integer number that exceeds either limit does not generate an illegal number; rather, it "wraps around".

Some IDL routines, such as the `plot` routine, ignore illegal numbers. Generally, however, you need a way to deal with these illegal numbers—for example, so that you can exclude them from subsequent mathematical operations. You do this with the `finite` command.

1.6. STRINGS

Strings store characters—letters, symbols, and numbers (but numbers as *characters*—you can't calculate with strings! A string constant such as **hello** consists of five letters. It takes 5 bytes to store this constant—one byte for each character. There are 256 possible characters for each of the bytes; with 2*26 letters (smalls and caps) and 10 digits, this leaves 104 other possibilities, which are used for things like semicolons and periods. You can generate an array of strings with **strarr** and a single string with **x = 'Hi there!!!'**.

Characters are represnted by byte numbers. You can easily find the byte number for a particular character in IDL; for example, to find the byte number that represents the semicolon character, type `print, byte(';')`. The answer: 59. Similarly, to find the string variable represnted by the byte number 59b, type `print, string(59b)`. Quick—what's the answer?

2. ORGANIZATIONAL STRUCTURES

2.1. SCALARS

A scalar is just a single number. For example, a string scalar is **joename= 'joe'**.

2.2. VECTORS

A vector is a one-dimensional array. For example, a three-element vector of names is **threenames = ['joe', 'ivan', 'mark']**.

2.3. ARRAYS

IDL handles arrays up to 8 dimensions, i.e. with 8 subscripts. Arrays with two subscripts can be mathematically treated a matrices using the # and ## operators, and various matrix manipulation routines; see IDL help under **matrices** and **matrix operators**. You create vectors and arrays using, for example, the **fltarr** or **findgen** commands (for floating point numbers; equivalent commands exists for all variable types). You populate them as appropriate, but try to avoid using for loops; instead, use **where**, appropriate use of the * operator, etc.

IDL provides a great deal of flexibility in using subscripts to address particular array elements, and this flexibility is what makes IDL so useful. For example, consider a two-dimensional array **a=findgen(100,100)**. Then:

```
b = a[ 23:25, 67:69]
```

makes **b** a 3×3 2-d array equal to a's array elements in the little box specified. The combination

```
indx = where( a gt 10.)
b = a[ indx]
```

makes **b** a 1-d array equal to the elements of **a** that are larger than 10. The combination

```
indx = where( a gt 10.)
jndx = where( a[ indx] le 100.)
b = a[ indx[ jndx]]
```

shows that you can subscript arrays with other arrays, and makes **b** equal to a 1-d array equal to the elements of **a** that are both larger than 10 and less than or equal to 100.

2.4. MATRICES

A matrix is just one step away from a 2-d array. IDL provides all of the standard, and many sophisticated and advanced matrix, operations. To multiply two matrices you use the # operator. Thus, the matrix product **C** of two matrices **A** and **B** is $\mathbf{C} = \mathbf{A} \# \# \mathbf{B}$—or $\mathbf{C} = \mathbf{A} \# \mathbf{B}$ depending on. . . a tricky little point about matrices having to do with row-major or column-major formats.

In a computer, a multidimensional data set can be indexed in two ways, the *column-major* and *row-major* formats. IDL uses the row-major format, as does Fortran; the other major language, C, uses column-major. Suppose you have a 2×2 matrix called **A**. In IDL's row-major format, when you type [*print, A*] IDL prints

$$\left[\begin{array}{cc} A_{0,0} & A_{1,0} \\ A_{0,1} & A_{1,1} \end{array} \right] , \tag{1a}$$

which is different from (i.e., it's the *transpose of*) what you are used to seeing in standard matrix notation which is the column-major format

$$\left[\begin{array}{cc} A_{0,0} & A_{0,1} \\ A_{1,0} & A_{1,1} \end{array} \right] . \tag{1b}$$

We use the row-major convention such that when displayed in a standard IDL *print* statement, they look correct. That means that our definition is the transpose of the standard one.

There are some matrix operations for which the difference is important. This includes not only multiplication, but also some other operations such as **invert** and **svsol**. IDL almost always assumes that the inputs to these other operations follow our row-major convention.

If you want to be a purist and define the matrices in the standard column-major manner, then go ahead and do so. You then need to do three things. First, if you want to see the matrix displayed in the usual way, then print its transpose by typing [*print, transpose(A)*]. Second, in all our IDL matrix equations, replace ## by #. Third, check any IDL procedure having a matrix as input to see what it assumes (The default is almost always row-major).

To be specific: if you follow our row-major convention, which is the transpose of the standard one, then the matrix product must be written

$$\mathbf{C} = \mathbf{A} \ \#\# \ \mathbf{B} \tag{2a}$$

while, in contrast, if you follow the standard column-major one then you must write

$$\mathbf{C} = \mathbf{A} \ \# \ \mathbf{B} \tag{2b}$$

Why does IDL do this nonstandard thing? It's because it's more straightforward for image processing, in which traditionally the images are scanned row-by-row (as in a TV set) instead of column-by-column. And IDL's origins are image processing, not matrix math. You might find IDL's convention annoying when you're doing matrix math, but this is more than compensated for by the intuitive feel you gain when doing image processing.

2.5. STRUCTURES

Structures are immensely useful for any project in which data of different types are related. For example, if you have a catalog of stars with positions and reddenings, you can put the whole catalog in a structure array in which each element of the array contains many quantities such as the name and position. And you can have arrays of structures. Structures allow you to create and customize your own data base. Having done this, using the **where** command allows you flexible access to anything with a one-line command.

We refer you to Chapter 7 of *Building IDL Applications* for a complete discussion of structures. Here we provide a quick example. There are two types of structure, *anonymous* and *named*. In our experience, anonymous structures are usually preferred.

2.5.1. Anonymous Structures

Our abovementioned star catalog has several different data elements. We group them all into a *structure*; each data element is called a *tag*. Let's call the structure A. Then we define the structure tags with the statement

```
A = {name: 'alpha ori', ra:5.3345, dec:-7.6568,reddening:fltarr(12)}
```

Now if you type `help,/struct,a` or `help,/st,a`, you will see on the screen

```
** Structure <752268>, 4 tags, length=72, data length=72, refs=1:
   NAME            STRING    'alpha ori'
   RA              FLOAT           5.33450
   DEC             FLOAT          -7.65680
   REDDENING       FLOAT     Array[12]
```

The number `<752268>` is arbitrary and will change from one anonymous structure to the next. The fact that a *number* appears instead of a name means that a is an *anonymous* structure. It has four tags: the name, the two positions, and 12 different measurements of reddening. In IDL you refer to a particular tag, e.g. *dec*, by typing `a.dec`:

```
print, a.dec
```

You could change things by typing

```
a.dec = 5.5
a.reddening[3] = 0.7
```

Or you could could populate the 12 reddening measurements by typing, for example,

```
a.reddening = [1.2, 1.4, 1.3, 1.6, 1.3, 1.4, 1.3, 1.3, 1.6, 1.3, 1.4, 1.3,]
```

You can redefine the tag names and number of tags in an anonymous structure—but not for a named structure. . .

2.5.2. Named Structures

You can also create a **named structure** by including the name (here the name is “star”):

```
b = {star, name: 'alpha ori', ra:5.3345, dec:-7.6568, reddening:fltarr(12)}
```

Now if you type **help, b, /str** you see

```
** Structure STAR, 4 tags, length=72, data length=72:
   NAME            STRING    'alpha ori'
   RA              FLOAT           5.33450
   DEC             FLOAT          -7.65680
   REDDENING       FLOAT     Array[12]
```

This says that the structure B is a named structure with name defined as *star*. We defined it to have the same tags as our anonymous structure A above. In contrast to an anonymous structure, *Once a named structure is defined, you cannot change the names of the tags or the number of tags.*

2.5.3. Arrays of Structures

If this were all, then structures wouldn't be very useful because you have observed 585 stars, say, and you'd need a separate structure for each. But you can create *arrays of structures*, e.g. for an anonymous structure we create the array *cataloga*

```
cataloga = replicate( a, 585)
```

and for our named structure we create the array *catalogb*

```
catalogb = replicate( {star}, 585) [OR catalogb = replicate( b, 585) ]
```

These statements create structure arrays of 585 elements. If you type `help, cataloga` IDL tells you this fact by printing on the screen

```
CATALOGA        STRUCT    = -> <Anonymous> Array[585]
```

and if you add the `structure` keyword (typing `help, cataloga, /st`) you get, as above,

```
** Structure <752268>, 4 tags, length=72, data length=72, refs=2:
   NAME            STRING    'alpha ori'
   RA              FLOAT           5.33450
   DEC             FLOAT          -7.65680
   REDDENING       FLOAT     Array[12]
```

You refer to particular elements of the array in the usual way, i.e. with an index number, but the index number comes directly after the array name, like this:

```
print, catalog[ 3].ra
print, catalog[3].reddening[5]
cataloga[ 3] = a
```

Or you can set values within a structure, for example

```
cataloga[ 3].name = 'carl heiles'
```

Try using structures when taking data for your experiments. You'll grow to love them!

3. APPENDIX: HOW FLOATING-POINT NUMBERS ARE STORED IN COMPUTERS

3.1. The 32-bit IEEE standard

Here we describe the 32-bit word case and assume the IEEE convention; see *Numerical Recipes, §20.1*, for more complete information. 24 bits are reserved for the mantissa and 8 for the exponent. Each reserves one bit for the sign, leaving 23 and 7 bits for numbers.

1. For the exponent, the the maxima and minimum numbers are $\pm 2^7 = \pm 128$—or more accurately, –128 to +127. This exponent applies to a binary number; converting to decimal, we have $2^{128} \approx 1.7 \times 10^{38}$. Roughly, this is the maximum number that can be stored as a single-precision float.

2. For the mantissa, the maximum and minimum numbers are $\pm 2^{23} \approx 10^{7}$. This means that the fractional error in a stored number is about 10^{-7}. As you do more and more calculations, using the results of one in the next, this fractional error increasees.

3. For double precision, the maximum number is about 10^{308} and the precision is about 10^{-16}.

3.2. The routine `machar`

The IDL routine `machar`, derived from the *Numerical Recipes* routine of the same name, returns a structure that contains all the above information.