ESCI 386 – IDL Programming for Advanced Earth Sciences Applications
Lesson 2 – Variables

Reading: An Introduction to Programming with IDL, Chapters 5 and 6

VARIABLE ASSIGNMENTS

- IDL uses dynamic variable typing, which means you do not have to declare variable types in advance.
- To assign a value to a variable you just use an equals sign.
- Rules for variable names:
  - Must begin with a letter.
  - Can be no longer than 128 characters.
  - Can include the 26 alphabetical characters, the digits 0 – 9, and the underscore character.
  - Cannot include spaces, hyphens, or any other character not listed above.
  - Variable names are case insensitive.
- IDL has certain reserved words that should never be used as variable names (see Table A.1 in Appendix A of textbook).
- Good programming practice:
  - Use lowercase letters for variable names, unless it makes good sense to use uppercase (e.g., T for temperature).
  - Make variable names descriptive of what they represent, but don’t make them too long.
  - Use an underscore to separate words in compound variable names.
  - Some examples of descriptive variables
    - gravity for gravity
    - rho or density for density
    - drop_volume for droplet volume
    - theta or pot_temp for potential temperature
    - T_celsius for Celsius temperature

GLOBAL VARIABLES AND SYSTEM VARIABLES

- A global variable is a variable whose value can be read at any level of the program or IDL session.
• *Global variables* are denoted by having an exclamation mark “!” as the first character of the variable name.

• A global variable that is read-only is called a *system variable*.

• IDL has several very useful system variables. A few that you should be aware of now are shown below

<table>
<thead>
<tr>
<th>Variable</th>
<th>Meaning</th>
<th>Example usage</th>
</tr>
</thead>
<tbody>
<tr>
<td>!dtor</td>
<td>Conversion from degrees to radians</td>
<td>radian = !dtor*degree</td>
</tr>
<tr>
<td>!radeg</td>
<td>Conversion from radians to degrees</td>
<td>degree = !rtod*radian</td>
</tr>
<tr>
<td>!pi</td>
<td>Single precision value of ( \pi )</td>
<td>circum = 2.0*!pi*radius</td>
</tr>
<tr>
<td>!dpi</td>
<td>Double precision value of ( \pi )</td>
<td>circum = 2.0*!dpi*radius</td>
</tr>
</tbody>
</table>

• Users can also create their own system variables using the `defsysv` procedure (such as was done in the “startup.pro” file.)

### INTEGER DATA TYPES

• IDL has several different integer data types. Some are signed, and some are unsigned.

• **IMPORTANT NOTE ON INTEGER DIVISION!**
  - Whenever two integers are divided in IDL (and most computer languages), the result is always an integer!
    - The expression 4/3 in IDL will be evaluated as 1, not 1.3333333.
  - You should get into the programming habit of never dividing by an integer unless you want the answer truncated. For the example above, you should type 4.0/3.0 instead.

• **byte data type**
  - byte data consists of unsigned integers from 0 to 255.
  - It is useful for image data using a 256-color gray shade scale.
  - To explicitly assign a value to a byte variable you can either:
    - append a “B” after the digits, such as “x = 15B”.
    - use the `byte()` conversion function, such as “x = byte(15)”.
o Assigning a value greater than 255 to a byte variable results in the values “wrapping” upward from zero.
  ▪ For example, “byte(256) = 0”, “byte(257) = 1”, etc.

o Assigning a negative value to a byte variable results in the values “wrapping” downward from 255.
  ▪ For example, “byte(-1) = 255”, “byte(-2) = 254”, etc.

• int data type

  o int data is also called short-integer data.
  o int data consists of signed integers ranging from −32,768 to 32,767.
  o To explicitly assign a value to an int variable you can either:
    ▪ append an “S” after the digits, such as “x = 15S”.
    ▪ use the fix() conversion function, such as “x = int(15)”.
  o If you haven’t set up the “startup.pro” file as suggested in Lesson 1, then int is the default data type for integers.

• uint data type

  o uint data consists of unsigned integers from 0 to 65,535.
  o To explicitly assign a value to a uint variable you can either:
    ▪ append a “U” after the digits, such as “x = 15U”.
    ▪ use the uint() conversion function, such as “x = uint(15)”.
  o Assigning a value greater than 65,535 to a uint variable results in the values “wrapping” upward from zero.
  o Assigning a negative value to a uint variable results in the values “wrapping” downward from 65,535.

• long data type

  o long data is also called long-integer data.
  o long data consists of signed integers ranging from −2^{31} to 2^{31} − 1.
  o Since we set up the “startup.pro” file to use 4-Byte integers by default, the long data type is the default for integers.
    ▪ Therefore, anytime we assign an integer to a variable without any appendage, IDL will assume it is of the long data type.
• However, you could also
  • append an “L” after the digits, such as “\(x = 15L\)”.
  • use the long() conversion function, such as “\(x = \text{long}(15)\)”.
• If you haven’t set up the “startup.pro” file as suggested in Lesson 1, then int is the default data type for integers.

• ulong data type
  o ulong data consists of unsigned integers from 0 to \(2^{32} - 1\).
  o To explicitly assign a value to a ulong variable you can either:
    ▪ append a “UL” after the digits, such as “\(x = 15UL\)”.
    ▪ use the ulong() conversion function, such as “\(x = \text{ulong}(15)\)”.
  o Assigning a value greater than \(2^{32} - 1\) to a ulong variable results in the values “wrapping” upward from zero.
  o Assigning a negative value to a ulong variable results in the values “wrapping” downward from \(2^{32} - 1\).

• long64 data type
  o long64 data consists of signed integers ranging from \(-2^{63}\) to \(2^{63} - 1\).
  o To explicitly assign a value to an long64 variable you can either:
    • append an “LL” after the digits, such as “\(x = 15LL\)”.
    • use the long64() conversion function, such as “\(x = \text{long64}(15)\)”.

• ulong64 data type
  o ulong64 data consists of unsigned integers from 0 to \(2^{64} - 1\).
  o To explicitly assign a value to a ulong64 variable you can either:
    ▪ append a “ULL” after the digits, such as “\(x = 15ULL\)”.
    ▪ use the ulong64() conversion function, such as “\(x = \text{ulong64}(15)\)”.
  o Assigning a value greater than \(2^{64} - 1\) to a ulong64 variable results in the values “wrapping” upward from zero.
  o Assigning a negative value to a ulong64 variable results in the values “wrapping” downward from \(2^{64} - 1\).
A FINAL NOTE ABOUT INTEGER DATA TYPES

- For most of the programming you will do with IDL you won’t really have to give much thought to which integer data type you are using.
- We’ve set our default to long, and so anytime we use an integer without explicitly specifying the type, that’s what it will be.
- It is worth mentioning one more time the warning about integer division, so here it is:
  - Whenever two integers are divided in IDL (and most computer languages), the result is always an integer!

FLOATING-POINT DATA TYPES

- IDL has two types of floating-point data:
  - Single-precision – This is the default, and can represent numbers whose absolute values are of the order of magnitude \(10^{-38}\) to \(10^{38}\).
    - It may be possible to represent numbers smaller than this. For instance, on our system I’ve seen IDL represent numbers as small as \(1.4 \times 10^{-45}\), but I wouldn’t trust it to do this reliably.
  - Double-precision - Can represent numbers whose absolute values are of the order of magnitude \(10^{-308}\) to \(10^{308}\).
    - It may be possible to represent numbers smaller than this. For instance, on our system I’ve seen IDL represent double-precision numbers as small as \(5.0 \times 10^{-324}\), but I wouldn’t trust it to do this reliably.
- IDL has several built-in features for converting floating-point numbers to integers.
  - fix() or long() functions – These truncate (remove all decimal places) with no rounding.
  - round() function – This rounds to the nearest integer.
  - floor() function – This returns closest integer that is less than or equal to the floating-point number.
  - ceil() function – This returns closest integer that is greater than or equal to the floating-point number.
Inf and NaN

- Unlike as in Fortran, if IDL calculates a floating-point number that is greater than the upper range that it can represent it doesn’t crash. Instead, it assigns the value Inf (which stand for infinity), and proceeds on it’s merry way. This is nice, because maybe it is enough for your purposes just to know the number is large.
  - Inf is assigned to any single-precision floating-point number that is larger than the upper-range that can be represented.
  - Infinity is assigned to any double-precision floating-point number that is larger than the upper-range that can be represented.
  - The result of dividing by 0.0 in IDL is Inf.

- For floating-point operations that mathematically yield undefined values, IDL assigns the value of NaN, which stands for (not-a-number).

- You can initialize single- and double-precision floating-point numbers to Inf, Infinity, or NaN by using the f_infinity, f_nan, d_infinity, and d_nan functions.

LOGICAL DATA TYPE

- IDL does not have an intrinsic logical (Boolean) data type.

- In evaluations where values are compared, a byte value of 1 is returned for true, and a byte value of 0 is returned for false.

STRING DATA TYPE

- String data and operations will be covered in a separate lesson.

CONVERSION OF DATA TYPES

- Conversion of one data type to another is very simple. With the exception of the int data type, the function to convert to a particular data type is simply that data type name.
  - For example, In order to convert to a floating-point variable, use the float() function.

- For the int data type, the conversion function is fix().
• The round(), floor(), and ceil() functions also convert floating-point data types to int or long, depending on the default for your session.

EXERCISES

1. Use the IDL statement print, byte(number) to convert the integers listed below into byte values. Make sure you understand why you get the values that are returned.
   a. 254
   b. 255
   c. 256
   d. 257
   e. −1
   f. −2

2. Type the following two lines into IDL and view the output (note that Line 2 actually contains three separate IDL statements.) Notice what happens as the value of gets smaller than \(10^{-38}\).
   Line 1: \(a = 1.0e-30\)
   Line 2: for i = 1, 20 do begin & a = a/10.0 & print, a & end

3. Do the same as in Exercise 2, only make the first line read: \(a = 1.0d-308\).

4. Type the following two lines into IDL and view the output (note that Line 2 actually contains three separate IDL statements.)
   Line 1: \(a = 1.0e30\)
   Line 2: for i = 1, 20 do begin & a = a*10.0 & print, a & end

5. Do the same as in Exercise 4, only make the first line read: \(a = 1.0d300\).