Proposal for coding conventions for the Hansl scripting language

The Gretl Team

September 2020

% Proposal for coding conventions for the Hansl scripting language % The Gretl Team % September 2020

Any fool can write code that a computer can understand.

Good programmers write code that humans can understand — Martin Fowler

Introduction

This is a proposal for implementing coding conventions for the scripting language *Hansl* implemented by the open-source software package *Gretl*.

The purpose of this proposal is to foster a discussion for establishing some common rules to follow when publishing gretl code.

Readability matters

One of the key insights of Guido van Rossum (creator of the Python programming language) is that code is read much more often than it is written.

The guideline published here is intended to improve the readability of Gretl code and make it consistent across the wide spectrum of Gretl code.

As it is said in the PEP 8 style guide (https://www.python.org/dev/peps/pep-0008/):

"*A style guide is about consistency. Consistency with this style guide is important. Consistency within a project is more important. Consistency within one module or function is the most important.*"

Limits of a style guide

One important rule to follow is: do not break backwards compatibility just to comply with this style guide!

Naming conventions

90% of what is considered clean code is based on how well you name your variables, functions, etc. Names that are visible to the user as public parts of the API should follow conventions that reflect usage rather than implementation.

Naming styles

The table below outlines some of the common naming styles in Hansl code and when you should use them:

Type	Naming convention	Examples
Public Function	Use a lowercase word or words. Separate words by underscores to improve readability.	function, my_function
Private Function	Use an uppercase word or words. Separate words by underscores to improve readability.	Function, My_Function
Variable	Use a lowercase single letter, word, or words. Separate words with underscores to improve readability.	x, var, my_variable
Constant	Use an uppercase single letter, word, or words. Separate words with underscores to improve readability.	CONSTANT, MY_CONSTANT, MY_LONG_CONSTANT
Package	Use a short, lowercase word or words. Separate words with underscores to improve readability.	package, my_package
File Name	Use a short, lowercase word or words. Separate words with underscores to improve readability.	run.inp, my_gretl_file.inp

Choose names carefully

Robert C. Martin, author of the well-known book "Clean Code" lists some standards for naming which make code both readable and maintainable.

1. Choose descriptive and unambiguous names.
2. Make meaningful distinctions.
3. Use pronounceable names.
4. Use searchable names.
5. Replace magic numbers with named constants.
6. Avoid encodings. Don't append prefixes or type information.

Code layout

Blank lines

Surround top-level functions and classes with two blank lines. Functions should be fairly self-contained and handle separate functionality. It makes sense to put extra vertical space around them, so that it's clear they are separate:

```
function scalar function_one ()
    return 1
end function


function scalar function_two ()
    return 2
end function
```

Use blank lines sparingly inside functions to show clear steps. Sometimes, a complicated function has to complete several steps before the return statement. To help the reader understand the logic inside the function, it can be helpful to leave a blank line between each step. If the function returns a value, put a blank line before return statement.

```
function list drop_excess_X (list X)
    list effX = timevar(X)
    list M = dropcoll(effX)

    list D = effX - M
    if nelem(D) > 0
        printf "Variables %s dropped for collinearity\n", varname(D)
    endif

    return M
end function
```

If you use vertical whitespace carefully, it can greatly improved the readability of your code. It helps the reader visually understand how your code splits up into sections, and how those sections relate to one another.

Maximum line length and line breaking

We suggest lines should be limited to 80 characters. This is because it allows you to have multiple files open next to one another, while also avoiding line wrapping.

Function arguments

Gretl will assume line continuation within a function block when defining a function arguments:

```
function void foo (scalar k,
                   matrix m)
    return k
end function
```

Implied continuation

If it is impossible to use implied continuation, then you can use backslashes to break lines instead:

```
matrix m = some_very_long_function_name(input_one,\
    input_two, input_three)
```

Binary operators

If line breaking needs to occur around binary operators, like + and *, it should occur before the operator:

```
# Recommended
total = (first_variable \
         + second_variable \
         - third_variable)
```

Bundle definition

When defining a Gretl bundle object with multiple key-value statements, we recommend to make use of line breaks after each key-value pair for readability purpose:

```
# Recommended
bundle B = defbundle(\
  "key_1", 1,\
  "key_2", 2)
```

Whitespace in expressions and statements

Whitespace can be very helpful in expressions and statements when used properly.

Whitespace around binary operators

Surround the following binary operators with a single space on either side:

- Assignment operators (=, +=, -=, and so forth)
- Comparisons (==, !=, >, <. >=, <=)
- Booleans (&&, !=, ||)

The following is not acceptable:

```
# Definitely do not do this!
if x >5 and x% 2== 0
    print('x is larger than 5 and divisible by 2!')
endif
```

When to avoid adding whitespace

In some cases, adding whitespace can make code harder to read. Too much whitespace can make code overly sparse and difficult to follow. The following examples outline some cases where you should avoid adding whitespace.

Immediately inside parentheses, brackets, or braces:

```
# Recommended
my_matrix = {1, 2, 3}

# Not recommended
my_matrix = { 1, 2, 3 }
```

Before a comma, semicolon, or colon:

```
# Recommended
strings S = deffarray("foo", "fufu"}

# Not recommended
strings S = deffarray( "foo" , "fufu"}
```

To align assignment operators:

```
# Recommended
var1 = 5
var2 = 6
some_long_var = 7

# Not recommended
var1          = 5
var2          = 6
some_long_var = 7
```

Comments

You should use comments to document code as it's written. It is important to document your code so that you, and any collaborators, can understand it. Here are some key points to remember when adding comments to your code:

- Use complete sentences, starting with a capital letter.
- Make sure to update comments if you change your code.

Block comments

Use block comments to document a small section of code. They are useful when you have to write several lines of code to perform a single action, such as importing data from a file.

We provide the following rules for writing block comments:

- Indent block comments to the same level as the code they describe.
- Start each line with a # followed by a single space.
- Separate paragraphs by a line containing a single #.

Inline comments

Inline comments explain a single statement in a piece of code. They are useful to remind you, or explain to others, why a certain line of code is necessary. The rules are as follows:

- Use inline comments sparingly.
- Write inline comments on the same line as the statement they refer to.
- Separate inline comments by two or more spaces from the statement.
- Start inline comments with a # and a single space, like block comments.
- Don't use them to explain the obvious.

Here is an example of an inline comment:

```
x = 5  # This is an inline comment
```

Documentation strings

Documentation strings, or docstrings, are strings enclosed in a `/*` and `*/` block. You can use them to explain and document a specific block of code.

The most important rules applying to docstrings are the following:

- Surround docstrings with `/*` and `*/` as in /* This is a docstring */.
- Write them for all public functions.
- Put the `*/` that ends a multiline docstring on the last line of the docstring.

```
function void foo()
    /* Solve quadratic equation via the quadratic formula.

    A quadratic equation has the following form:
    ax**2 + bx + c = 0

    There always two solutions to a quadratic equation: x_1 and x_2. */

    # do something
end function
```

When to ignore these coding guidelines

The short answer to this question is never. The coding guidelines make sure that you'll have clean, professional, and readable code. This will benefit you as well as collaborators and potential employers.

However, some guidelines are inconvenient in the following instances:

- If complying with our coding guidelines would break compatibility with existing software.
- If code surrounding what you're working on is inconsistent with our coding guidelines.
- If code needs to remain compatible with older versions of Gretl.