Writing Scripts in Gretl

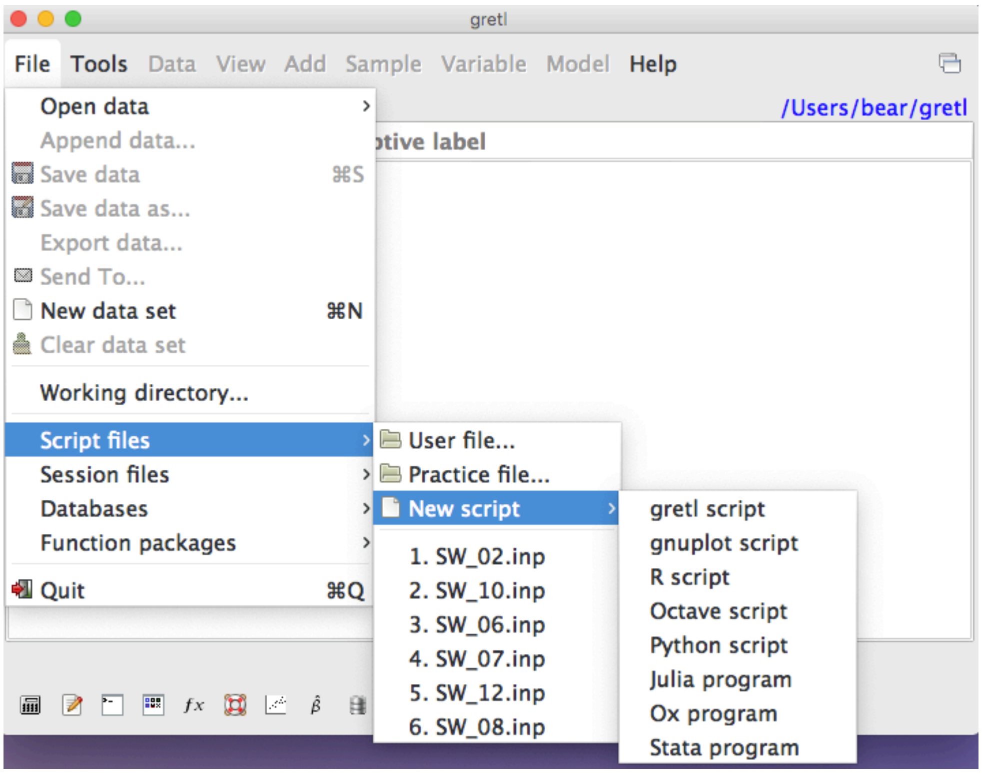
Working with scripts

So far we have done several exercises in gretl by clicking the menu. This mode of working is easy and intuitive, but not productive when you want to complete some complicated operations which are time consuming and may include repetitions, and impossible to record what you have done. Here we introduce the scripting mode, e.g., writing commands into a program. The resulting program is also called a *script* or *source* file, with which you can save the process of your analysis instead of results. This enables you to review and revise your analysis, and helps in the communication between collaborators.

Open, edit, and save a script file

To start working with scripts, we need to open a new blank script file. This is done by choosing

> File > Script files > New script > gretl script

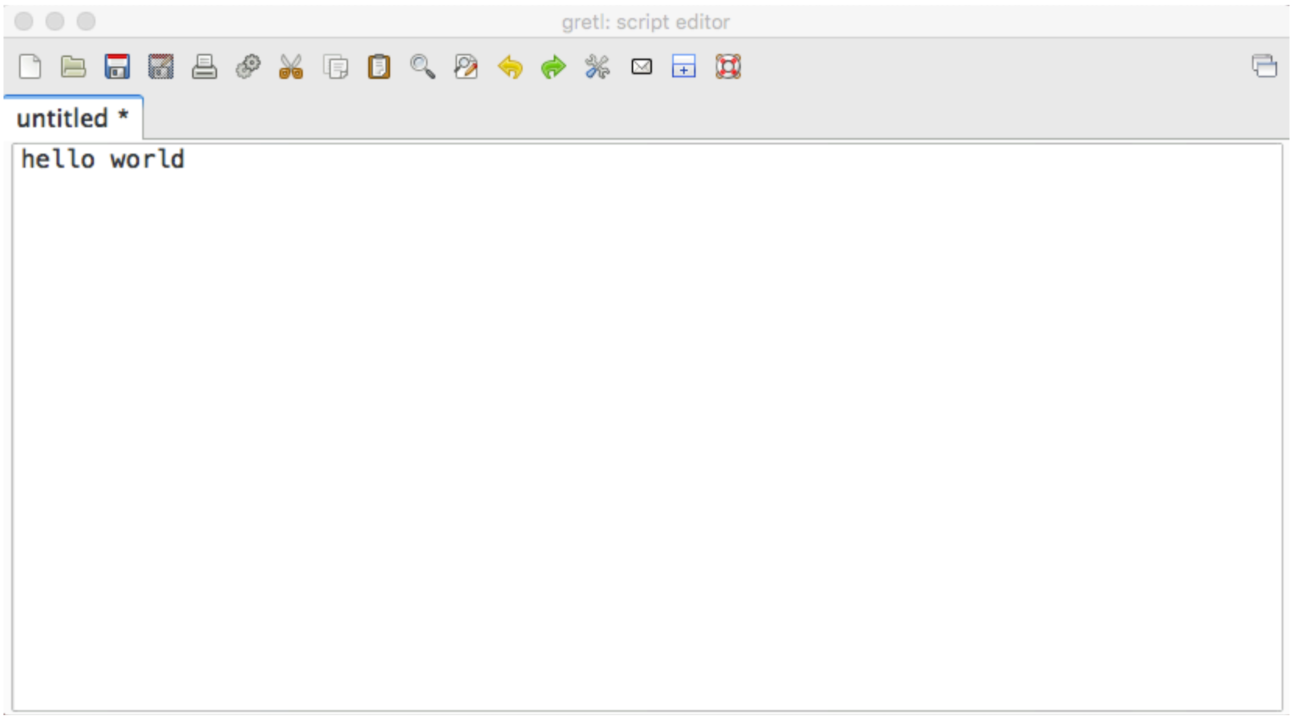


A new editor window appears, and you can edit (type something) and save by clicking the third icon on the menu bar. The default location of gretl scripts is the working directory, which by default is set as

```
…/user/gretl/
```

where `user` is your username of the OS. You can save your scripts elsewhere, as well as change the default working directory into any location by choosing

> File > Working directory…



```
hello world
```

Sample scripts

There are many built-in sample scripts in gretl for study purpose. You can find them from

> File > Script files > Practice file…

For example, the file "ps2-1" under "Ranamathan" contains the following commands:

```
# PS2.1 using DATA2-1, illustrating frequency distributions, Section 2.1
open data2-1
help freq
# Because SHAZAM and gretl have different command structures, the outputs
# don't correspond exactly
freq vsat
freq msat
```

Colors are automatically assigned. Lines starting with a # is usually called *comments*, which are not executable. With comment lines you can write explanations about the program for the ones who may read including yourself. Other lines are executable, where words in boldface are *commands*. The purpose of this sample program is obvious.

You can execute the entire program by clicking the third icon ⚙ on the menu bar. To run specific lines, you can select them with mouse and right-click, then choose "Run" (in your own program this appears as "Execute region").

What can we do?

In scripting mode we can do almost everything that can be done from the menu, and even more. Though it is free to write your own program, the basic structure of a program may include the following building blocks.

Preamble	Special settings of this program.

Data import	Reading data from a file or a database.
Data manipulation	Data cleaning, data transformation, defining new variables, etc.
Descriptive statistics	Summary statistics, plots, etc.
Main analysis	Model fitting, testing, prediction, etc.
Representation of results	Tabulation, graphing, etc.
Export	Save important results into separate files.

The programing language used in gretl is called `hansl`. The "Hansl primer" from the Help menu provides a very nice introduction. You are strongly recommended to learn this material by typing down every example in it.

Exercise. Type the following program into a script file and execute line-wise. Learn the commands in the program with Command reference in the Help menu.

```
# Import built-in data file "data3-1"
open data3-1

# Show descriptive statistics
summary price sqft

# Draw a scatter plot of price against sqft
gnuplot price sqft --output=display

# Run regression of price on sqft with a constant term
ols price const sqft
```

Exercise. Try the following script and think about why it is written in this way. Use command reference to see the meaning of `meantest`.

```
# one sample t-test using two sample t-test with unequal variances
nulldata 20
series X = seq(1,20)
series Y = 0
meantest X Y --unequal-vars
series Z = 8
meantest X Z --unequal-vars
```

Programming

Programming is the logic and art of writing a program. For a high level programming language such as hansl (or Python, R, etc.), we need at least to know how to save values into memory (data type), how to structure operations (control flow), and how to define new commands (user-written functions). Here we introduce the first two.

Basic data types

The most basic data type in gretl are `series` and `scalar`. A series is a variable, e.g., a set of observations in a sample, whereas a scalar is a single value. Suppose we have imported the data file "data3-1". We can create a new variable named `lsqft` defined as the log of `sqft`. This can be done by

```
series lsqft = sqft^2
```

The following script save the sample mean of price into `mprice`.

```
scalar mprice = mean(price)
```

Here, `mean()` is a *function*. The difference between a command and a function is that a function takes input variables but a command does not. More functions can be found from the Function reference in the Help menu.

A `list` is a set of series, and a `string` saves a sentence of natural languages such as English enclosed by double quotes "".

```
list X = const sqft
ols price X
string greeting = "Hello, world!"
print greeting
```

The `if` statement — conditional executions

Knowing commands and functions is not enough for writing good programs. In addition, you need to know how to control the flow of executions in order to make complicated calculations. The `if` statement makes it possible to execute different commands in different situations. For example, suppose you have a variable "height" indicating the height of students, and a variable "sex" indication the sex of students. You know that there is a common measurement error such that the true value of height for male students should be the current observation plus 1, and the true value for female students should be the current observation minus 2. In order to correct this error, *for each student*, you can do the followings.

```
if [this student is a male]
        do [trueheight = height + 1]
else
        do [trueheight = height - 2]
end
```

Note that the above lines are pseudo-codes. In gretl, the general form of `if` statement is

```
if condition1
    commands1
elif condition2
    commands2
else
    commands3
endif
```

The `conditions` in the `if` statement need to be Boolean operations (intuitively, questions that can be answered with yes or no). The `elif` and `else` parts are optional. The pseudo-codes above can be written as follows (not executable yet).

```
if sex[i] == 1 # if male
    trueheight[i] = height[i] + 1
else # if female
    trueheight[i] = height[i] – 2
endif
```

The `loop` statement — repeated executions

In the above example the `if` statement need to be executed for each student (each observation of the sample data). This kind of repetition is made possible by the `loop` statement. The idea here is to do something for each member belonging to a set, where the set can be a series, a list, or some other data types. There are many types of loops, for details you are refereed to Chapter 12 of the *Gretl's User's Guide* or Chapter 8 of *A Hansl Primer*. Here we introduce the index loop. The general form of an index loop is

```
loop counter=min..max
    commands
endloop
```

The `counter` in this expression can be any word, and usually we use `i`, `k`, or `j`. The `min` and `max` are integers which specify the range of `counter`. That is, the value of `counter` is initially set as `min`, then increased by one in each step, until it reaches `max`, such as `i=1..10`.

Try the following script (for the above example) and understand how an index loop works.

```
nulldata 20
series height = randgen(N, 165, 10) # generate height from N(165, 10^2)
series sex = randgen(B, 0.5, 1) # generate sex from Binom(p=0.5, n=1), male=1
series trueheight

loop i=1..$nobs # $nobs returns the number of observations in the sample
    if sex[i] == 1 # if male
        trueheight[i] = height[i] + 1
    else # if female
        trueheight[i] = height[i] - 2
    endif
endloop
```

Practice

The Lucas numbers

The Lucas numbers are defined by the recurrence equation $L_n = L_{n-1} + L_{n-2}$ with $L_1 = 1$ and $L_2 = 3$. Do the followings in script mode.

1. Create a null dataset with 25 observations.
2. Create a variable "L" which stores the first 25 Lucas numbers.
3. Create another variable "ratio" which is defined by $\text{ratio}_i = L_i / L_{i-1}$ for $i = 2, \ldots, n$ with $\text{ratio}_1 = 1$.

Observe how the values of "ratio" changes as *n* increases.

A sample script

```
nulldata 25
series L
series ratio

L[1] = 1
L[2] = 3
ratio[1] = 1
ratio[2] = L[2] / L[1]
loop i=3..$nobs --quiet
    L[i] = L[i-1] + L[i-2]
    ratio[i] = L[i] / L[i-1]
endloop
```