The Stata Journal (2010)
10, Number 2, pp. 281–296

Speaking Stata: Finding variables

Nicholas J. Cox
Department of Geography
Durham University
Durham City, UK
n.j.cox@durham.ac.uk

Abstract. A standard task in data management is producing a list of variable names showing which variables have specific properties, such as being of string type, or having value labels attached, or having a date format. A first-principles strategy is to loop over variables, checking one by one which have the property or properties concerned. I discuss this strategy in detail with a variety of examples. A canned alternative is offered by the official command `ds` or by a new command, `findname`, published formally with this column.

Keywords: dm0048, findname, ds, variable names, variable labels, value labels, types, formats, characteristics, data management

1 Introduction

In early versions of Stata, users were typically dealing with at most tens of variables. In recent versions, users may be dealing with a hundred or even a thousand times more. Such a difference can turn some data-management tasks that are trivial in principle into labors that are substantial in practice, unless you have the right tools. This column focuses on a common need: finding which variables possess some specified property, such as being a string variable, having value labels attached, or having a date format. The common feature is needing a list of variable names that can be used in some later commands.

The problem is approached on two levels, a first-principles, or low-level, approach and a higher-level approach centered on a new command, `findname`, which is published formally with this article. The deeper principles are more important than any particular implementation: no command can cover all imaginable needs, and there will always be tasks that require a more fundamental approach.

The main difficulty in this problem is not that Stata will refuse to provide the information you need to answer the questions you have. On the contrary, information of all kinds on your observations and on the basic structure and extra properties of your dataset is typically only a command away. What can create difficulties, however, is that you can be swamped with information you do not need or presented with information in a form you cannot easily reuse, at least without retyping a lot of variable names. Such retyping is clearly unattractive, tedious, and error-prone.

A simple example illustrates the main dilemma. Say you notice that the string variables in your dataset make your Stata life more difficult by messing up calculations

or making them impossible. Some systematic treatment of your string variables is called for, perhaps applying `encode` or `destring` (see [D] **encode** or [D] **destring**) or perhaps moving them to one end of the dataset with `order` (see [D] **order**). The first need is to get the names of the variables concerned, which are likely to be scattered throughout the dataset. Perhaps the data came with one or more string identifiers, which may be among the first few variables in the dataset. But, equally likely in any substantial project, all sorts of extra string variables may have been created in approaching some problem and so are scattered through the dataset.

A few sessions with Stata will show users at least one way of approaching this problem. `describe` (see [D] **describe**) displays among other things the storage type of each variable, which for a string variable will be one of the `str` types. But no one wants to scroll through what may be screenfuls of `describe` output. The same applies to `codebook` (see [D] **codebook**) output, to mention another way to find out about the dataset. There should be better ways, and this column explains what they are.

2 The main idea

2.1 varlists: Lists of variable names

One of the first pieces of Stata jargon that users encounter is the term *varlist*. A varlist is a list of variable names. If you read in the canonical Stata `auto.dta` by typing `sysuse auto`, then `price mpg rep78` is a varlist applying to your data, because each of the variables named is part of that dataset. So also is `mpg make`, and so also is any other list containing one or more variable names. Getting varlists that name precisely the variables we care about—no more, no fewer—is thus the problem here.

We can easily imagine typing all the variable names for `auto.dta`, but the idea is not appealing, and early Stata teaching should cover abbreviated ways of catching some of or all the variable names. The simplest and most general abbreviation (often called a wildcard) is `*`, which catches all the variable names in the current dataset.

2.2 Looping over variables

A simple way to see which variables satisfy some property is just to loop over variables, examining each variable in turn. Spelling this out in a kind of pseudocode,

```
1. initialize list of desired variables to empty

2. for each variable that exists {
        2a. check whether it has the desired property
        2b. if it has {
                add name of variable to the list
            }
}
```

The basic idea, labeled 2, is that of a loop over variables with exactly the same structure as you would use for any number of mundane tasks—throwing out magazines you no longer need, deciding whether to attend concerts or movies in a program, or whatever it is.

The prerequisite step, labeled 1, is important too. If the place you store the list already holds stuff from some other task, then you may get slightly confused or even make incorrect decisions, just as you might if you added this week's shopping list to last week's.

Another common twist on the problem is that the complementary list may be useful too. In this example, a list of numeric variables is just as useful as a list of string variables. We will discuss complements later in the column.

The first part of the problem is thus how to loop. `foreach` (see [P] **foreach**) is the most general looping command in Stata and as such the key to finding variables. A basic tutorial was given in a previous column (Cox 2002) and so an introduction is not repeated here. That tutorial explains the basics of local macros, which appear in many later examples.

Focusing on `foreach` does not exclude the possibility of approaching problems with `forvalues` (see [P] **forvalues**), which in special cases could be simple and attractive. For example, if your variables had the names `var1`–`var42`, then `forvalues` would be an alternative to `foreach`. But this kind of situation is exceptional.

```
foreach v of var * {
        commands go here
}
```

is a loop over all variables. `var` is short for `varlist` and `*`, as mentioned earlier, includes all variables.

2.3 How to check properties: Extended macro functions

The other part of the problem is checking variable properties. Let us read in `auto.dta` and return to the example of finding string variables. In this case, it is easy to see (say, by using `describe`) that there is just one string variable, `make`. But how would we find that out automatically?

Frequently, a specific tool for solving your problem is given by a so-called extended macro function, documented in [P] **macro**. Several tools are documented under this heading. Thus the storage type of a variable in `auto.dta` is returned by the construct `: type`. This can be used to put the type in a local macro:

```
. local t : type make
```

or we could cut out the macro and just `display` the information in question:

```
. display "`: type make´"
str18
```

This kind of shortcut is also documented in [P] **macro**. Had the variable type been `str7` or `str42`, the storage type would have been reported accordingly. We have thus a criterion for a string variable: that the first three characters of the type would be `str`. A check of possible storage types at (say) `help data types` shows that the numeric types are `byte`, `int`, `float`, `long`, and `double`, so there is no possibility of confusion if we use this criterion. Given the `substr()` function for extracting substrings, some code for this problem is

```
foreach v of var * {
        local t : type `v´
        if substr("`t´", 1, 3) == "str" {
             local strvarlist `strvarlist´ `v´
        }
}
```

So within the loop, we look up the type for each variable. If the first three characters are `str`, we add the name of the variable in question to our list. The key line,

```
local strvarlist `strvarlist´ `v´
```

defines the local macro `strvarlist` to be whatever it was before the line was entered, together with the new name, from the local macro named `v`.

This code can be refined in two ways. First, the presumption here is that the local macro `strvarlist` does not exist before the loop; otherwise put, just before the loop, `strvarlist` is empty. In careful code, we would ensure that is true by blanking out the macro in advance. Second, using another local macro (here `t`) with the loop is not essential. As in the `display` example earlier in this section, we could code without it.

```
local strvarlist
foreach v of var * {
        if substr("`: type `v´´", 1, 3) == "str" {
                local strvarlist `strvarlist´ `v´
        }
}
```

If we run this code on `auto.dta`, we get the one string variable, `make`.

```
. display "`strvarlist´"
make
```

2.4 How to check properties: confirm and capture

Another commonly used command for this kind of problem is `confirm` (see [P] **confirm**). `confirm` appears to offer a more straightforward answer to this particular question. Typing

```
. confirm str var make
```

is answered by silence. Stata's way of confirming what is asserted to be true is to assent tacitly: no news is good news, in plainer terms. If you follow that command with a check on a numeric variable, you get not silence, but an error message.

```
. confirm str var price
´price´ found where string variable expected
r(7);
```

So `confirm` is like a fire or burglar alarm: it goes off if something is wrong. That alarm would stop any loop over variables as soon as an error was first encountered. But often we want to keep on going, which is where `capture` (see [P] **capture**) enters the story.

```
. capture confirm str var price
. display _rc
7
```

These commands eat the error message, while the nonzero return code that is diagnostic of an error remains defined, so we can have it both ways. We just need to check for a zero error code.

```
local strvarlist
foreach v of var * {
        capture confirm str var `v´
        if _rc == 0 {
             local strvarlist `strvarlist´ `v´
        }
}
```

Note that we could go through the loop building two lists (or in other problems, even more lists). For example,

```
local strvarlist
local numvarlist
foreach v of var * {
        capture confirm str var `v´
        if _rc == 0 {
             local strvarlist `strvarlist´ `v´
        }
        else local numvarlist `numvarlist´ `v´
}
```

The decision-making is simple. If the error code is zero—because the variable is a string variable—we add its name to the list of string variables. If the error code is not zero—because the variable is not a string variable—that can only mean that it is a numeric variable, and we add its name to the list of numeric variables. This idea does not presuppose that there are variables of both kinds; if there are not, we will find that out from an empty list.

However, although it is possible to build two or more lists in this manner, that is not often necessary. Later in the column, we will see other ways to get lists through basic set operations such as complementation.

2.5 How to check properties: count and summarize

Two further commonly used commands for checking properties are `count` and `summarize` (see [D] **count** and [R] **summarize**). `summarize` is perhaps more likely to spring to users' minds, but many tasks that might be assigned to `summarize` are more simply tackled by `count` (Cox 2007a,b,d).

Consider the problem of finding variables with any missing values. Various commands, including `nmissing` (Cox 1999, 2001a, 2003a, 2005), offer ways to address this problem. Here we use our loop-and-check technique. The checking is just counting how many values are missing in each variable. If there is at least one, we add the variable to the list.

```
local missvarlist
quietly foreach v of var * {
        count if missing(`v´)
        if r(N) > 0 {
             local missvarlist `missvarlist´ `v´
        }
}
```

The `missing()` function is the best way to check for missings. Among other advantages, it covers both numeric and string variables (Rising 2010), so we do not need to check for numeric or string and then `count if 'v' >= .` or `count if 'v' == ""` accordingly. `count` is noisy by default; that is, the resulting count is displayed. Here `quietly` suppresses the display but the result stored in `r(N)` is inspected. (If `r(N)` is new to you, start by reading `help return`.) `quietly` could appear just on `count` or on a larger segment of code. The choice is not usually important and is typically a matter of convenience. I like my lines of code to remain short, and that preference sometimes drives small decisions on this point.

This little problem could be varied. Suppose the problem were to find variables not with any values missing, but with all values missing. If all values are missing, the count result `r(N)` will be equal to the number of observations `_N`, so all we need change is one line:

```
local missvarlist
quietly foreach v of var * {
        count if missing(`v´)
        if r(N) == _N {
             local missvarlist `missvarlist´ `v´
        }
}
```

There are yet other ways to do it. For example, you might type
`count if !missing('v')` and then check whether the resulting count was zero.

An approach with `count` will suffice for many basic queries. For example, zero or negative values may make matters difficult or impossible if a variable is necessarily defined as positive, or ideally occurs as positive only, as when, say, a logarithmic transform is being contemplated. More generally, we can count values lying inside or outside de-

fined intervals; the choice between counting "inside" and counting "outside" is usually one of convenience.

Sometimes you do need to reach for `summarize`. Suppose we wanted to find all variables that held proportions, fractions between 0 and 1. Storage type will not help here, because, although we must hold such variables as either `float` or `double`, that is not a criterion. But if we `summarize`, we can inspect minimum and maximum. One tip here is to use the `meanonly` option of `summarize` when possible. The name `meanonly` is misleading, because even under this option other summary statistics are produced (Cox 2007c). With this problem, we have a choice over handling string variables. It is not an error to feed a string variable to `summarize`, so we could just plow straight ahead:

```
local propvarlist
foreach v of var * {
        summarize `v´, meanonly
        if r(min) >= 0 & r(max) <= 1 {
             local propvarlist `propvarlist´ `v´
        }
}
```

After `summarize` with a string variable, a call to the results `r(min)` and `r(max)` yields missings, so no string variable will qualify. Because `summarize, meanonly` displays no results, we can remove the `quietly`, but it would do no harm if we left it.

Hang on, however: The minimum being at least 0 and the maximum being at most 1 includes all indicator or dummy variables coded 0 or 1. We might mind about that inclusion or we might not mind. If we do mind, we could think about adding a check in the code for variables that were only ever 0 or 1 (or missing), but now the code starts getting complicated. Let us pause for a moment and think about a different technique.

2.6 Set operations

Many problems of finding variable names can be phrased in set theory terminology as, for example, the intersection or union of two or more sets, or a subset, complement set, or set difference. So let us define proportions, in a strict sense, as the subset of numeric variables that do have nonmissing values in $(0, 1)$ but are not indicators (nonmissing values only 0 or 1). Let us now approach the problem in steps. First, we get the numeric variables using an approach similar to a previous example:

```
local numvarlist
foreach v of var * {
        capture confirm num var `v´
        if _rc == 0 {
             local numvarlist `numvarlist´ `v´
        }
}
```

Now we find the indicators. Let us say that an indicator is numeric, but all values are 0 or 1 or missing. Naturally, we start with `numvarlist`, not all variables (`*`).

```
local indvarlist
quietly foreach v of var `numvarlist´ {
        count if (`v´ == 0 | `v´ == 1 | missing(`v´))
        if r(N) == _N {
             local indvarlist `indvarlist´ `v´
        }
}
```

Now all we need to do is remove the members of `indvarlist` from `propvarlist` obtained earlier. [P] **macro lists** documents how to do such set manipulations. What we want is

```
local propvarlist : list propvarlist - indvarlist
```

We could choose to put all the code together into one loop over variables, and I sometimes do that. But you can end up with a tangle that is hard to maintain or debug or that is legal code but hides logical errors.

3 The findname command

3.1 Purpose of findname

At the time of writing, the official command that comes closest in spirit to the approach discussed so far is `ds`. For some purposes, `lookfor` (see [D] **lookfor**) also offers a quick and easy solution. However, neither `ds` nor `lookfor` nor any other command can offer canned one-line solutions to all the little problems above, which is precisely why knowing the approach from first principles is valuable.

`ds` and its relatives under differing names have had a long and complicated history in various official and user-written versions (for example, Anonymous [1992]; Cox [2000, 2001b]; Weiss [2008]). Although `ds` was for a while undocumented—meaning precisely, documented through a help file but not a manual entry—`ds` was restored to full status in the 9 March 2010 update to Stata 11. Be that as it may, this column offers yet another variation, `findname`, with extended functionality and a revised syntax.

`findname` lists variable names of the dataset currently in memory in a compact or detailed format and lets you specify subsets of variables to be listed, either by name or by properties (for example, the variables are numeric). In addition, `findname` leaves behind in `r(varlist)` the names of variables selected so that you can use them in a subsequent command.

If two or more options specifying properties of variables are specified, `findname` identifies only those variables that satisfy all the option specifications, that is, the intersection of all the subsets identified. Its `not` option provides a direct way to identify the complementary set. Two or more calls to `findname` with results saved in local macros using its `local()` option may be used together with macro operations to produce the union, set difference, etc., of different subsets.

3.2 Examples of findname

That is the executive summary, but now let us take it more slowly through examples.

`findname` can find all string or numeric variables through a call to its `type()` option:

```
. findname, type(string)
. edit `r(varlist)´
. findname, type(numeric)
. summarize `r(varlist)´
```

Note that the variable names not only are displayed but also are saved in `r(varlist)` so that we can use that, or more precisely its local macro persona, in a subsequent command. But watch out: `r()` results are ephemeral and often quickly overwritten. For example, the `summarize` call above destroys `r(varlist)` and leaves its own `r()` results behind instead. To avoid the annoying problem of losing the results you only just obtained, an easy alternative is to copy results immediately to a local macro with a name of your own choice:

```
. findname, type(string)
. local strvarlist `r(varlist)´
```

An even easier alternative is to call `findname`'s `local()` option:

```
. findname, type(string) local(strvarlist)
. edit `strvarlist´
```

The difference is that the local macro `strvarlist` will remain visible within the same session, program, or do-file, unless you yourself overwrite it. If it is really important, save the contents using `notes` (see [D] **notes**). A `local()` option is not included in `ds`.

To find proportion variables, we could type

```
. findname, all((@ >= 0 & @ <= 1) | missing(@)) local(propvarlist)
. findname, all(@ == 0 | @ == 1 | missing(@)), local(indvarlist)
. local propvarlist : list propvarlist - indvarlist
```

Options `all()` and `any()` are new to `findname` over `ds`. A call to `all()` or `any()` features `@` as a placeholder for variables, so each variable name is substituted in turn for `@`.

There is a twist in the first command just given. When we looped for ourselves, we could rely on the fact that `summarize` just ignores any missings. But `findname` does not do that. If it did, that would seem reasonable behavior up to the point when you wanted to find missings, when it would start to seem highly unreasonable. So in careful code, we need to be explicit about the possibility of missings existing in a variable.

(*Continued on next page*)

A `placeholder()` option is provided for some unlikely but not impossible situations. Your `@` key might be broken. Or you might need a string comparison based on the literal character `@`, such as when you want to look for variables containing that character, perhaps variables containing email addresses. `placeholder()` lets you specify an alternative, although in turn you need to choose something that you do not also need to use literally.

What is likely to be more important is that `findname` quietly traps all tests in `all()` or `any()` calls that result in a type mismatch. In this example, the implied

```
. count if varname >= 0 & varname <= 1
```

would fail as illegal if *varname* was a string variable, because double quotes would be needed around 0 and 1. However, `findname` is smart enough to catch such mismatches. (The smartness here is more a matter of brute force, but it works anyway.)

The `local()` option does come in especially handy in this example. Not only do the macrolist directives require macro names, but also the second call to `findname` overwrites the `r(varlist)` left in memory by the first.

Here is another way to approach this example. Again the results of one `findname` are fed to another, but this time the `not` option of `findname` is used to find the complement of the set of indicators within the larger set:

```
. findname, all((@ >= 0 & @ <= 1) | missing(@)) local(propvarlist)
. findname `propvarlist´, all(@ == 0 | @ == 1 | missing(@)) not
```

If logarithms were being contemplated, then

```
. findname, any(@ <= 0)
```

finds any numeric variables with zero or negative values.

A new example with `findname` is finding variables with only integer values. Storage type is again no criterion here, because nothing stops you holding integers in variables of `float` or `double` type. A suitable criterion is that no values are changed by rounding to the nearest integer, say,

```
. findname, all(@ == int(@))
```

Again any string variables are just ignored in a call like this. The `int()` function is used here, which always rounds toward zero such that `int(-1.2)` is -1, but the `floor()` or `ceil()` function would do just as well (Cox 2003b).

As a final example, which variables are constants, holding precisely the same value in all observations? Unless such variables had been constructed deliberately, say, for some graphical purpose, they would normally appear as candidates for `drop`ping. Here is a `findname` solution:

```
. findname, all(@ == @[1])
```

If all values are the same, then each value is the same as the first value, a criterion that applies to numeric and string variables alike. The first observation is not special in this problem (unless your dataset has only one observation, and if that were so, you would not be asking this question). If there were at least 7 observations, or at least 42, then the criteria `@ == @[7]` or `@ == @[42]` would work as well. And even without those restrictions, `@ == @[_N]` would always work. But the first solution given above is the simplest solution guaranteed to work.

For comparison, here is a loop-and-check solution:

```
local constvarlist
foreach v of var * {
        sort `v´
        if `v´[1] == `v´[_N] {
             local constvarlist `constvarlist´ `v´
        }
}
```

We must `sort` each variable to be sure of picking up any departures from constancy, because otherwise the first and last values might just be equal as a coincidence. To put it more positively, even if there are only two distinct values and one of those occurs just once, `sort`ing guarantees that we will spot that case, because the unique oddity will be sorted to one or the other end of the data.

More statistically minded solutions to this question are not so straightforward. You might think in terms of finding the minimum and maximum of each variable and checking whether they are the same. However, `summarize` ignores missing values; that is no solution for strings; and even if there were no missing values and no string variables, you still have to scan the output of `summarize` somehow.

3.3 Further features of findname

`findname` has a bundle of options for searching not only according to storage types, but also according to formats, label and characteristic names, and included text. Its syntax now distinguishes more clearly than `ds` between (for example) finding out whether value labels are attached, what they are named, and what is included in their text. The ability to search value-label text and characteristic text as well as names is new over `ds`.

Combination of criteria for variables is also extended over `ds`. You can combine options, searching for those variables with all of two or more criteria satisfied. The logic of `findname` is that of finding the intersection of sets. As we have seen, unions and set differences need two or more calls to `findname`, although the `not` option offers scope to get the complement of any specification.

(*Continued on next page*)

4 Syntax for findname

`findname` [*varlist*] [*if*] [*in*] [`,` `insensitive` `local(`*macname*`)` `not`
`placeholder(`*symbol*`)` `alpha` `detail` `indent(`#`)` `skip(`#`)` `varwidth(`#`)`
`type(`*typelist*`)` `all(`*condition*`)` `any(`*condition*`)` `format(`*patternlist*`)` `varlabel`
`varlabeltext(`*patternlist*`)` `vallabel` `vallabelname(`*patternlist*`)`
`vallabeltext(`*patternlist*`)` `char` `charname(`*patternlist*`)` `chartext(`*patternlist*`)`]

4.1 Definitions for options

typelist used in `type(`*typelist*`)` is a list of one or more types, each of which may be `numeric`, `string`, `byte`, `int`, `long`, `float`, or `double`, or may be a numlist, such as `1/8` to mean `str1` `str2` ... `str8`. Examples include

`type(int)`	is of type `int`
`type(byte int long)`	is of integer `type`
`type(numeric)`	is a numeric variable
`type(1/40)`	is `str1`, `str2`, ..., `str40`
`type(numeric 1/2)`	is numeric or `str1` or `str2`

patternlist used in, for example, `format(`*patternlist*`)`, is a list of one or more patterns. A pattern is the expected name or text with the likely addition of the characters `*` and `?`. `*` indicates 0 or more characters go here and `?` indicates exactly 1 character goes here. Examples include

`format(*f)`	format is `%`#`.`#`f`
`format(%t*)`	has time or date format
`format(%-*s)`	is a left-justified string
`varl(*weight*)`	variable label includes word `weight`
`varl(*weight* *Weight*)`	variable label includes word `weight` or `Weight`

To match a phrase, it is important to enclose the entire phrase in quotes.

`varl("*some phrase*")`	variable label has `some phrase`

If instead you used `varl(*some phrase*)`, then only variables having labels ending in `some` or starting with `phrase` would be listed.

condition used in `all()` or `any()` is a true-or-false condition defined by an expression in which variable names are represented by `@`. For example, `any(@ < 0)` selects numeric variables in which any values are negative.

4.2 Options

Control options

`insensitive` specifies that the matching of any pattern in *patternlist* be case-insensitive. For example, `varl(*weight*) inse` is an alternative to, and more inclusive than, `varl(*weight* *Weight*)`.

`local(`*macname*`)` puts the resulting list of variable names into local macro *macname*.

`not` specifies that *varlist* or the specifications given define the set of variables not to be listed. For instance, `findname pop*, not` specifies that all variables not starting with the letters `pop` be listed. The default is to list all the variables in the dataset or, if *varlist* or particular properties are specified, to list the variable names so defined.

`placeholder(`*symbol*`)` specifies an alternative to `@` to use in the `any()` or `all()` option. This should only be necessary for making string comparisons involving `@` as a literal character (or if your `@` key is somehow unavailable).

Display options

`alpha` specifies that the variable names be listed in alphabetical order.

`detail` specifies that detailed output identical to that of [D] **describe** be produced. If `detail` is specified, `indent()`, `skip()`, and `varwidth()` are ignored.

`indent(`#`)` specifies the amount the lines are indented.

`skip(`#`)` specifies the number of spaces between variable names; the default is `skip(2)`.

`varwidth(`#`)` specifies the display width of the variable names; the default is `varwidth(12)`.

Selection by data types, values, and formats

`type(`*typelist*`)` selects variables of the specified *typelist*. Typing `findname, type(string)` would list all the names of string variables in the dataset, and typing `findname pop*, type(string)` would list all the names of string variables beginning with the letters `pop`.

`all(`*condition*`)` selects variables that have all values satisfying *condition*. If either `if` or `in` is specified, attention is restricted to the observations specified.

`any(`*condition*`)` selects variables that have any values satisfying *condition*. If either `if` or `in` is specified, attention is restricted to the observations specified.

With either `all()` or `any()`, *condition*s that mismatch type are ignored.

`format(`*patternlist*`)` selects variables whose format matches any of the patterns in *patternlist*. `format(*f)` would select all variables with formats ending in `f`, which presumably would be all `%`#`.`#`f`, `%0`#`.`#`f`, and `%-`#`.`#`f` formats. `format(*f *fc)` would select all formats ending in `f` or `fc`.

Selection by variable and value labels

`varlabel` selects variables with defined variable labels.

`varlabeltext(`*patternlist*`)` selects variables with variable-label text matching any of the words or phrases in *patternlist*.

`vallabel` selects variables with defined value labels.

`vallabelname(`*patternlist*`)` selects variables with value-label names matching any of the words in *patternlist*.

`vallabeltext(`*patternlist*`)` selects variables with value-label text matching any of the words or phrases in *patternlist*. If either `if` or `in` is specified, attention is restricted to the observations specified. Either way, no attention is paid to value labels that do not correspond to values present in the data.

Selection by characteristics

`char` selects variables with defined characteristics. Notes in the sense of [D] **notes** are characteristics.

`charname(`*patternlist*`)` selects variables with characteristic names matching any of the words in *patternlist*.

`chartext(`*patternlist*`)` selects variables with characteristic text matching any of the words or phrases in *patternlist*.

5 Conclusions

Finding variable names within Stata datasets is a basic task that has increased in importance as the size of datasets allowed has increased. As always in data management, users need versatile basic commands for common versions of the problem and ways of going beyond those commands when they do not offer a solution. `findname` is offered here as an alternative to `ds`. The strategies and tricks in this column for looping and checking are offered as an alternative to both.

6 Acknowledgments

My earlier work on versions of `ds` was aided by suggestions from Richard Goldstein, William Gould, Jay Kaufman, and Fred Wolfe. More recently, Maarten Buis, Martin Weiss, Vince Wiggins, and several members of Statalist provided helpful comments both directly and indirectly that led to the development of `findname`.

7 References

Anonymous. 1992. dm67.1: Short describes, finding variables, and codebooks. *Stata Technical Bulletin* 8: 3–5. Reprinted in *Stata Technical Bulletin Reprints*, vol. 2, pp. 11–14. College Station, TX: Stata Press.

Cox, N. J. 1999. dm67: Numbers of missing and present values. *Stata Technical Bulletin* 49: 7–8. Reprinted in *Stata Technical Bulletin Reprints*, vol. 9, pp. 26–27. College Station, TX: Stata Press.

———. 2000. dm78: Describing variables in memory. *Stata Technical Bulletin* 56: 2–4. Reprinted in *Stata Technical Bulletin Reprints*, vol. 10, pp. 15–17. College Station, TX: Stata Press.

———. 2001a. dm67.1: Enhancements to numbers of missing and present values. *Stata Technical Bulletin* 60: 2–3. Reprinted in *Stata Technical Bulletin Reprints*, vol. 10, pp. 7–9. College Station, TX: Stata Press.

———. 2001b. dm78.1: Describing variables in memory: update to Stata 7. *Stata Technical Bulletin* 60: 3. Reprinted in *Stata Technical Bulletin Reprints*, vol. 10, p. 17. College Station, TX: Stata Press.

———. 2002. Speaking Stata: How to face lists with fortitude. *Stata Journal* 2: 202–222.

———. 2003a. Software update for nmissing and npresent. *Stata Journal* 3: 349.

———. 2003b. Stata tip 2: Building with floors and ceilings. *Stata Journal* 3: 446–447.

———. 2005. Software update for nmissing and npresent. *Stata Journal* 5: 607.

———. 2007a. Speaking Stata: Counting groups, especially panels. *Stata Journal* 7: 571–581.

———. 2007b. Speaking Stata: Making it count. *Stata Journal* 7: 117–130.

———. 2007c. Stata tip 50: Efficient use of summarize. *Stata Journal* 7: 438–439.

———. 2007d. Stata tip 51: Events in intervals. *Stata Journal* 7: 440–443.

Rising, B. 2010. Stata tip 86: The missing() function. *Stata Journal* 10: 303–304.

Weiss, M. 2008. Stata tip 66: ds—A hidden gem. *Stata Journal* 8: 448–449.

About the author

Nicholas Cox is a statistically minded geographer at Durham University. He contributes talks, postings, FAQs, and programs to the Stata user community. He has also coauthored 15 commands in official Stata. He wrote several inserts in the *Stata Technical Bulletin* and is an editor of the *Stata Journal*.