Session 3: Consistency Checks

At the end of this lesson participants will be able to:

- Use the command ***errmsg*** to display messages to the interviewer
- Use ***errmsg*** with ***select*** keyword
- Use ***if then else*** statements to implement consistency checks across multiple variables
- Implement both "hard" and "soft" edit checks
- Create and run test plans for consistency checks
- Implement consistency checks on dates
- Declare and use logic variables

Consistency Checks with Two Variables

Up to now, we have been able to limit the responses for each variable to only a set of valid responses using value sets. What if we want to check for consistency between two different variables?

For example, let's prevent the interviewer from entering more bedrooms than there are rooms in the house. Click on the NUMBER_OF_BEDROOMS field in the forms tree and then click on *Logic* in the toolbar to open the logic editor. Add the following code in the procedure for `NUMBER_OF_BEDROOMS`:

```
PROC NUMBER_OF_BEDROOMS

// Ensure that number of bedrooms is not more than number of rooms.
if NUMBER_OF_BEDROOMS > NUMBER_OF_ROOMS then
    errmsg("Number of bedrooms cannot exceed number of rooms");
    reenter;
endif;
```

The ***if*** statement only executes the code between the ***then*** and then ***endif*** if the condition (NUMBER_OF_BEDROOMS > NUMBER_OF_ROOMS) is true. The ***errmsg*** statement will display a message to the user. The ***reenter*** statement forces the interviewer to stay in the current field and does not allow them to advance to the next field.

Note that we put our logic in the proc for NUMBER_OF_BEDROOMS and not in the proc for NUMBER_OF_ROOMS. This is because when we are in the proc for NUMBER_OF_ROOMS, the value for NUMBER_OF_BEDROOMS has not yet been entered. When creating consistency checks we always put the logic for the check in the proc for the *last* field involved in the check.

Let's take a look at another example, the edit specifications call for the following control:

- If RELATIONSHIP is spouse, then MARITAL_STATUS should be coded 1 (never married), 2 (divorced) or 3 (widowed).

Which proc should we put this check in? Since MARITAL_STATUS comes after RELATIONSHIP we will put in the proc for MARITAL_STATUS. For starters let's just show an error if the RELATIONSHIP is spouse and MARITAL_STATUS is single:

```
PROC MARITAL_STATUS

// Ensure that spouse is not single
if MARITAL_STATUS = 1 and RELATIONSHIP = 2 then
    errmsg("Marital status of spouse cannot be single");
    reenter;
endif;
```

Here we see that it is possible to combine multiple conditions using *and*. This condition will only be true if both subconditions are true. It is also possible to combine conditions using *or* which will be true if either the first subcondition or the second subcondition is true. For example, if we expand our check to include spouses who are single, divorced or widowed we could write:

```
PROC MARITAL_STATUS

// Ensure that spouse is not single, divorced or widowed
if (MARITAL_STATUS = 1 or MARITAL_STATUS = 3 or MARITAL_STATUS = 4) and
RELATIONSHIP = 2 then
    errmsg("Marital status of spouse cannot be single, divorced or widowed");
    reenter;
endif;
```

Note that when combining multiple *and* and *or* expressions, *and* expressions are evaluated first and then *or* expressions are evaluated which can sometimes lead to unexpected results. You can use parentheses to force the order of evaluation you want like we do above. What happens to the above expression without the parentheses if the relationship is not 2 and the marital status is 3?

Of course there are a couple of simpler ways to write this check without using or. We can use "<>" which means "not equal to":

```
// Ensure that spouse is not single
if MARITAL_STATUS <> 2 and RELATIONSHIP = 2 then
    errmsg("Marital status of spouse cannot be single, divorced or widowed");
    reenter;
endif;
```

Or we can use the keyword *in*:

```
// Ensure that spouse is not single
if MARITAL_STATUS in 1,3,4 and RELATIONSHIP = 2 then
    errmsg("Marital status of spouse cannot be single, divorced or widowed");
    reenter;
endif;
```

The in operator will be true if the value matches any of the numbers in the comma separated list. It also supports ranges by separating the start and end of the range with a colon (:). For example *BEDROOMS in 1:4* will be true if BEDROOMS is 1,2,3 or 4.

There is also a **NOT** operator. The **NOT** operator negates an expression: turning true values into false and false into true. We can use that to simplify the above expression:

```
// Ensure that spouse is NOT single
// If RELATIONSHIP is spouse then MARITAL_STATUS must not be never married.

if RELATIONSHIP = 2 and MARITAL_STATUS = 1 then
    errmsg("Spouse cannot be never married.");
    reenter;
endif;
```

We can also write this as:

```
if RELATIONSHIP = 2 and NOT MARITAL_STATUS in 2:4 then
    errmsg("Spouse cannot be never married.");
    reenter;
endif;
```

Better Error Messages

We can improve our error message by adding parameters to the string. Let's go back to the check on the number of bedrooms and display the number of rooms and the number of bedrooms in the message:

```
PROC NUMBER_OF_BEDROOMS

// Ensure that number of bedrooms does not exceed number of rooms.
if NUMBER_OF_BEDROOMS > NUMBER_OF_ROOMS then
    errmsg("Number of bedrooms, %d, exceeds number of rooms, %d.",
           NUMBER_OF_BEDROOMS, NUMBER_OF_ROOMS);
    reenter;
endif;
```

The "**%d**"s in the message are replaced by the values of the variables that follow in the order that they are listed.

Let's do the same for the check on marital status and relationship:

```
// Ensure that spouse is not single
if MARITAL_STATUS in 1,3,4 and RELATIONSHIP = 2 then
    errmsg("Relationship is spouse and marital status is %d. Spouse cannot be
single, divorced or widowed.", MARITAL_STATUS);
    reenter;
endif;
```

It would be better if we could include the name of the person in the error message as an additional clue to the interviewer**.** Since the name is an alpha variable we use **%s** instead of **%d.** *(%d is for decimal value.)*

```
PROC MARITAL_STATUS

// Ensure that spouse is not single
if MARITAL_STATUS in 1,3,4 and RELATIONSHIP = 2 then
    errmsg("%s has relationship spouse and marital status %d. Spouse cannot
be single, divorced or widowed.",
           NAME(curocc()), MARITAL_STATUS);
    reenter;
endif;
```

Why is there a whole bunch of extra space after the name in the error message? Remember that alpha variables in the dictionary are fixed length. This means that they get padded with blank spaces. We can remove the trailing blank spaces by using the ***strip()*** function.

```
PROC MARITAL_STATUS

// Ensure that spouse is not single
if MARITAL_STATUS in 1,3,4 and RELATIONSHIP = 2 then
    errmsg("%s has relationship spouse and marital status %d. Spouse cannot
be single, divorced or widowed.",
           strip(NAME(curocc())), MARITAL_STATUS);
    reenter;
endif;
```

Finally, rather than show the numeric value of the marital status it would be nicer to show the label from the value set. We can do this using the function ***getlabel()***. ***getlabel()*** returns the value set as an alpha value. It takes the name of the variable (or value set) and the value to use. In this case, both the variable and the value are MARITAL_STATUS. Since ***getlabel()*** returns an alpha we need to change the %d to a %s.

```
PROC MARITAL_STATUS

// Ensure that spouse is not single
if MARITAL_STATUS in 1,3,4 and RELATIONSHIP = 2 then
    errmsg("%s has relationship spouse and marital status %s. Spouse cannot
be single, divorced or widowed.",
           strip(NAME(curocc())), getlabel(MARITAL_STATUS, MARITAL_STATUS));
    reenter;
endif;
```

Errmsg with select

If we add select to the ***errmsg*** we can give the user the option of going back to correct either field. We can use this in our bedrooms example:

```
PROC NUMBER_OF_BEDROOMS

// Ensure that number of bedrooms does not exceed number of rooms.
if NUMBER_OF_BEDROOMS > NUMBER_OF_ROOMS then
    errmsg("Number of bedrooms, %d, exceeds number of rooms, %d.",
           NUMBER_OF_BEDROOMS, NUMBER_OF_ROOMS)
           select("Fix number of rooms", ROOMS,
                  "Fix number of bedrooms", NUMBER_OF_BEDROOMS);
endif;
```

With the ***select*** we no longer need the ***reenter*** since CSEntry will automatically reenter the field that the interviewer selects.

Note that the ***select*** clause is part of the ***errmsg*** statement so there is no semicolon in between the ***select*** and the ***errmsg***.

If you want to allow the user to ignore the error and enter the next field, you can add an option to the select statement with the keyword ***continue***:

```
PROC NUMBER_OF_BEDROOMS

// Ensure that number of bedrooms does not exceed number of rooms.
if NUMBER_OF_BEDROOMS > NUMBER_OF_ROOMS then
    errmsg("The number of bedrooms, %d, is greater than the number of rooms,
%d.",
           NUMBER_OF_BEDROOMS, NUMBER_OF_ROOMS)
           select("Fix number of rooms", ROOMS,
                  "Fix number of bedrooms", NUMBER_OF_BEDROOMS,
                  "Ignore error", continue);
endif;
```

Now the message dialog will have a third button labelled “ignore” that, when clicked, will move to the next field, in this case to type of main dwelling.

This is commonly referred to as a “soft edit check” as opposed to the previous “hard edit check” that does not allow the interviewer to move on until the inconsistency is fixed. The advantage of a soft edit check is that the interviewer won’t get stuck, however, data quality may suffer. In a CAPI census soft edit checks are generally preferred in order to prevent interviewers from getting stuck and to keep interview times to a minimum.

Let's add a select to the relationship-marital status check too:

```
// Ensure that spouse is not single
if MARITAL_STATUS in 1,3,4 and RELATIONSHIP = 2 then
    errmsg("%s has relationship spouse and marital status %s. Spouse cannot
be single, divorced or widowed.",
           strip(NAME(curocc())), getlabel(MARITAL_STATUS, MARITAL_STATUS))
    select("Fix marital status", MARITAL_STATUS,
           "Fix relationship", RELATIONSHIP);
endif;
```

Testing Consistency Checks

When testing consistency checks involving multiple variables it is important to test all possible combinations of the variables. To do this we can create a test matrix. For example, to test all possible combinations of our consistency check between MARITAL_STATUS and RELATIONSHIP we would use the following matrix that shows the expected result for each combination of the variables involved:

	Marital Status			
Relationship	Never married (1)	Married (2)	Divorced (3)	Widowed (4)
Spouse (2)	Error	OK	Error	Error
Not-spouse (<> 2)	OK	OK	OK	OK

Using this matrix, you can test each of the 8 possible combinations and make sure that you get the expected result. This ensures that all possible cases are tested.

By combining the test matrices from all the consistency checks in the application, you can create a test plan for the survey/census application to verify the application works correctly when changes are made. Often times changes to one part of the questionnaire can have unintended effects on skips and consistency checks in other parts of the questionnaire so it is important to have a test plan that is used after every set of changes to ensure that no problems are introduced.

Calculations

In question D02 the respondent gives the number of boys living in the household, girls living in the household and the total number of children living in the household. Let's check that the sum of the girls and the boys is equal to the total.

```
if BOYS_IN_HOUSEHOLD + GIRLS_IN_HOUSEHOLD <> CHILDREN_IN_HOUSEHOLD then
    errmsg("Boys (%d) plus girls (%d) does not match total (%d).",
           BOYS_IN_HOUSEHOLD, GIRLS_IN_HOUSEHOLD, CHILDREN_IN_HOUSEHOLD )
        select("Correct boys", BOYS_IN_HOUSEHOLD,
               "Correct girls", GIRLS_IN_HOUSEHOLD,
               "Correct total", CHILDREN_IN_HOUSEHOLD);
endif;
```

CSPro logic supports the following mathematical operators:

Addition	+
Subtraction	-
Multiplication	*
Division	/
Modulo (remainder)	%
Exponentiation	^

Checking Dates

Let's add a consistency check between the date of birth and the age. We can use the function datediff() to calculate the age based on the interview date and the date of birth:

```
PROC DAY_OF_BIRTH

// Ensure that age matches date of birth
if datediff(DATE_OF_BIRTH, DATE_OF_INTERVIEW, "y") <> AGE then
    errmsg("Age (%d) does not match date of birth (%d)", AGE, DATE_OF_BIRTH)
        select("Correct Age", AGE, "Correct date of birth", DATE_OF_BIRTH);
endif;
```

This seems to work correctly for a household with only one member but it fails if you have two or more members. Let's use errmsg to display the date of birth to try see what is going on. This is a common technique for debugging CSPro logic.

```
PROC DAY_OF_BIRTH

errmsg("Date of birth is (%d)", DATE_OF_BIRTH);
```

When we run this on a household with a few members and we are on the first line of the roster, the DATE_OF_BIRTH being reported matches the DATE_OF_BIRTH from the last line. Why? Normally in the proc for a variable in a roster, CSPro can infer the occurrence number for the variables in the roster by looking what row number the program is currently on. So when on the first row of the roster CSPro turns DATE_OF_BIRTH into DATE_OF_BIRTH(1) and on the second row it uses DATE_OF_BIRTH(2). However, in this case DATE_OF_BIRTH is not in the roster, only its subitems are. Since DATE_OF_BIRTH is not in the roster, CSPro doesn't know what occurrence number to use so it ends up using the last one. We can fix this by adding a subscript when we refer to DATE_OF_BIRTH. What function can use to get the current row number?

```
PROC DAY_OF_BIRTH

// Ensure that age matches date of birth
if datediff(DATE_OF_BIRTH(curocc()), DATE_OF_INTERVIEW, "y") <> AGE then
    errmsg("Age (%d) does not match date of birth (%d)", AGE, DATE_OF_BIRTH)
        select("Correct Age", AGE, "Correct date of birth", DATE_OF_BIRTH);
endif;
```

This works but it would be better if the error message informed the interviewer what the date of birth that we calculated was.

```
PROC DAY_OF_BIRTH

// Ensure that age matches date of birth
if datediff(DATE_OF_BIRTH(curocc()), DATE_OF_INTERVIEW, "y") <> AGE then
    errmsg("Age (%d) does not match date of birth (%d). Based on date of
birth age should be %d",
        AGE, DATE_OF_BIRTH(curocc()), datediff(DATE_OF_BIRTH(curocc()),
        DATE_OF_INTERVIEW, "y"))
        select("Correct Age", AGE, "Correct date of birth", YEAR_OF_BIRTH);
endif;
```

We should also handle the case where the age or date of birth is unknown.

```
PROC DAY_OF_BIRTH

// Ensure that age matches date of birth
if AGE <> 999 and DATE_OF_BIRTH(curocc()) <> 99999999 then
    if datediff(DATE_OF_BIRTH(curocc()), DATE_OF_INTERVIEW, "y") <> AGE then
        errmsg("Age (%d) does not match date of birth (%d). Based on date of
birth age should be %d",
            AGE, DATE_OF_BIRTH(curocc()),
            datediff(DATE_OF_BIRTH(curocc()), DATE_OF_INTERVIEW, "y"))
            select("Correct Age", AGE, "Correct date of birth",
                   YEAR_OF_BIRTH);
    endif;
endif;
```

Logic Variables

It would better if we didn't repeat the datediff calculation twice. Repeating code makes it harder to maintain. We make fix a bug in one copy of the code but forget to do so another. To avoid repeating ourselves we can declare a logic variable to hold the value of our calculation. Logic variables are like dictionary variables but are only for use in calculations in logic and don't get shown on forms or saved to the data file. To create a variable, we declare it as follows:

```
numeric aNumber;
string anAlphanumeric;
alpha(20) anAlphaNumericWithAFixedLength;
```

Numeric variables hold numbers (including numbers with fractional parts) and string variables hold alphanumeric values. Alpha variables are like strings but with a fixed length. In early versions of CSPro string variables didn't exist and it was common to use alpha variables. In modern CSPro you should always use string variables instead of fixed length alpha variables.

Coding style

We recommend using mixed case (a.k.a. camelCase) for logic variables in order to distinguish them from dictionary variables.

We can declare a variable to hold the result of the datediff as follows:

```
numeric calculatedAge;
calculatedAge = datediff(DATE_OF_BIRTH(curocc()), DATE_OF_INTERVIEW, "y");
```

Now we can use that variable in place of the datediff:

```
if calculatedAge <> AGE then
    errmsg("Age (%d) does not match date of birth (%d). Based on date of
birth age should be %d",
        AGE, DATE_OF_BIRTH(curocc()), calculatedAge)
        select("Correct Age", AGE, "Correct date of birth", YEAR_OF_BIRTH);
endif;
```

It is possible to combine the declaration and the initialization of the variable into a single line:

```
numeric calculatedAge = datediff(DATE_OF_BIRTH(curocc()), DATE_OF_INTERVIEW,
"y");
```

When to Use Consistency Checks

It is tempting to place consistency checks on every question in your application in order to maximize data quality but it is important to realize that every check you add comes at a price. Beyond the time it takes to implement and properly test each check, you also need to consider the additional time required to train and support a more complex application. Every check you add increases the risk that there will be bugs in your application when it gets to the field. Unlike a key from paper application, a bug in the field can mean an enumerator being stuck and abandoning an entire questionnaire. For surveys, you generally have a small sample and better trained enumerators so this is less of a concern. However, for a census, enumerators are not as experienced and supporting them in the field is much more of a challenge. In a census, due to the number of respondents, you can use imputation to correct many inconsistencies with minimal impact on the overall results. In any case, just the fact that the CAPI instrument imposes range checks and skip patterns will be a significant improvement in data quality over a paper census. For a CAPI census application, we recommend only adding consistency checks on the key demographic fields in the household roster and leaving potential inconsistencies in other parts

of the questionnaire to be corrected post data collection. In addition, we recommend using mainly "soft" edit checks so that the enumerator is never blocked by the application.

Exercises

Implement the following consistency checks from the edit specifications. Test that they are correct. Create informative error messages. Use test matrices to assure you test all possibilities.

1. Display an error message if the head of household is less than 15 years old.

2. Check that B17 (age at first marriage) is less than or equal to the current age (B05).

3. Display an error message if the highest level of education (C02) is university or graduate but the age is less than eighteen. Make this a "soft check" so that the interviewer can ignore the error if they wish.

4. Ensure that if the highest level of education (C02) is "standard 5" or higher, then the individual knows how to read or write (C03). Make this a soft check.

5. Add a consistency check in section F that displays an error message **if** the household has a flush toilet in question F10 **and** does NOT also specify piped water inside house for question F11. Use a select clause to allow them to correct either F10 or F11.

6. Ensure that the total number of children in D03 is equal to the sum of the boys and the girls. Do the same for question D04.

7. Check that the date that the death occurred in question E05 is within the last five years, since the section should only include deaths occurring in that period. Use the interview date to calculate the number of years since the death occurred. For example, the interview date is 20016-10-05 and the date of death is 2016-09 then you should an error message. You can use datediff to calculate the difference between the interview date and the date of death as we did for the age and date of birth check, however you will need to construct a full date from just the month and year of death. Since no day is given use 1 as the day (assume first day of the month). Note that datediff rounds to the nearest year so to do the check correctly you will need to get the difference in months and compare it to 60 (5 years times 12 months).