[2]p()*1/2-^ [1]p1-^ TJ
3 10000pt

0.00.5em

0.0.00.5em

0.0.0.00.5em

0.0.0.0.00.5em

0.0.0.0.0.00.5em
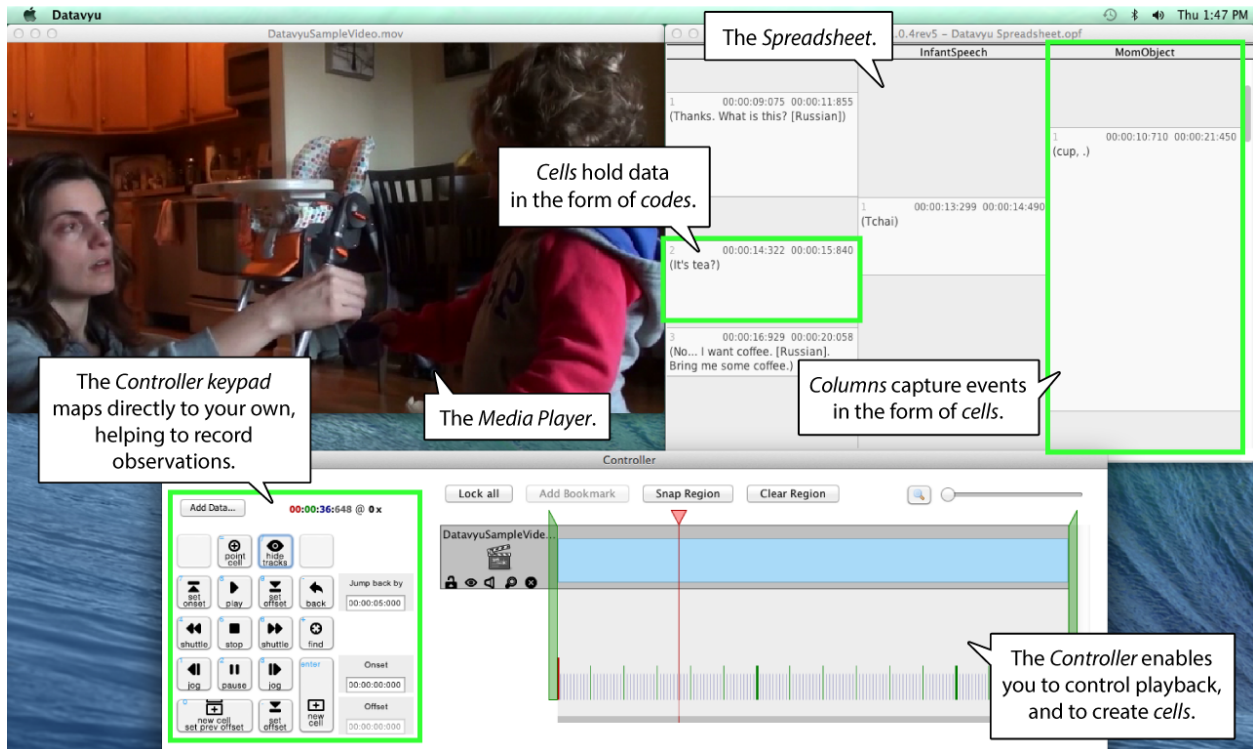
Datavyu Documentation
Release 1.3

Databrary Project

Aug 30, 2022

Contents

Datavyu is a complete software package for visualizing and coding behavioral observations from video data sources. Designed by - and for - behavioral scientists, Datavyu facilitates data coding and sharing through the ongoing Databrary data library project.

Note: Datavyu is an open source software package. You should familiarize yourself with previous version's release notes, to be aware of each version's features and potential issues. If you encounter a bug, you can report it and get help by posting to the support forum.

In addition to the powerful software package, Datavyu provides a Ruby-based API to help you automate common tasks and ensure accuracy in your data. The API greatly enhances the Datavyu experience, and is well worth the effort of learning some programming fundamentals.

When you are ready to start using Datavyu, the following chapters outline everything you need to know to become an expert Datavyu user, from first principles to advanced strategies.

Chapter 1

Software Guide

Datavyu is a Java-based application that runs on Windows, Mac, and on Linux operating systems. Its primary goal is to link behavioral researchers with their video and to provide a way for researchers to record their observations, extract their data for analysis, and share their work. Using a spreadsheet template that specifies which categories of events you're interested in coding, Datavyu enables researchers to record events and build on prior analyses.

This guide provides all the information you need to start using Datavyu, with installation instructions, an in-depth description of the software components, and a series of tutorials that target common operations. The reference section lists terms and definitions, as well as detailing any file format requirements and limitations.

1.1 Installation

Datavyu is a Java-based application, which is easy to install on Mac OS X and Windows. The following sections describe Datavyu's system requirements and then the installation process itself.

1.1.1 Requirements

Hardware Requirements

The hardware requirements listed below are the minimum needed for Datavyu. More RAM or a faster processor will improve Datavyu's performance, especially when working with higher-resolution data sources.

- 1GB of RAM.

- 1.2 GHz processor.

You will also need enough disk space for your videos and a keypad, either as part of your keyboard, or as an external device.

Software Requirements

Datavyu requires Java 1.6 or higher. Many operating systems include Java by default, but if you have *not* installed Java, you will need to do so.

The newest Mac build now has a built in version of Java. It no longer matters what version of Java is on your computer because Datavyu will use the one that it installs.

Datavyu supports video playback through Quicktime.

Note: If you are using Quicktime 7.7.5 or later, you will need to simply custom install Quicktime to include Java libraries in your version. The screenshot shows just how easy it is! Older versions of Quicktime can be typically installed. If you are on Windows 10, please download QuickTime version 7.7.6 here. Later versions of QuickTime will not work on Windows 10.
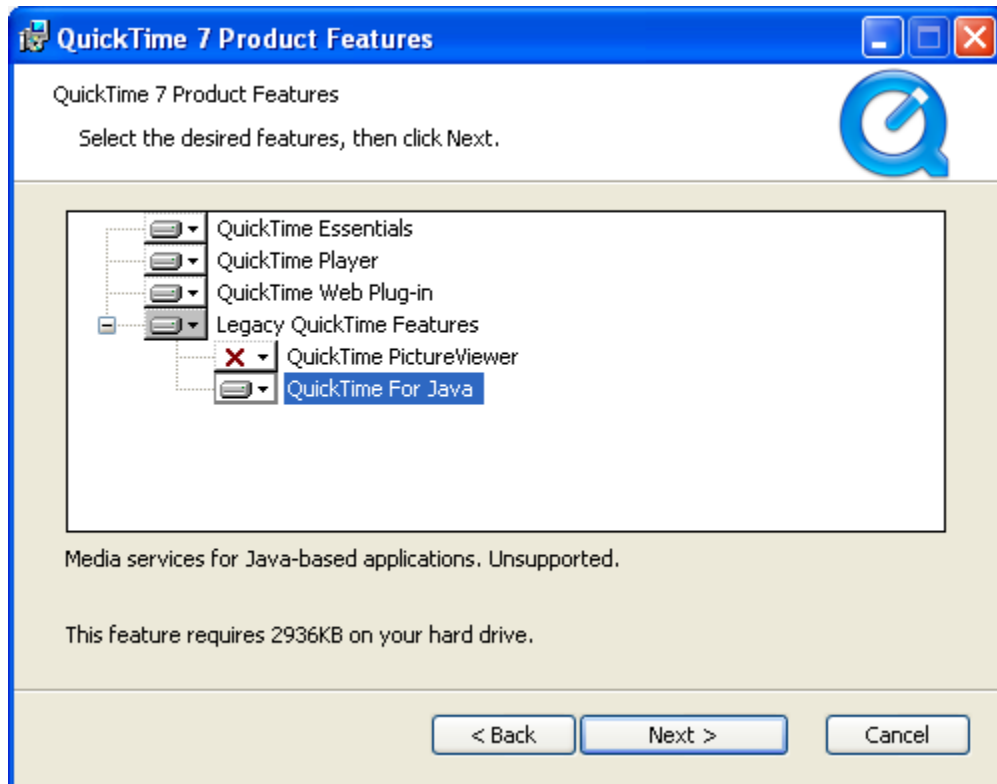

1.1.2 Install Datavyu

Mac OS X

1. Before downloading Datavyu, please be sure that your security settings allow the installation of non-Mac applications. To change this permanently or temporarily go to System Preferences > Security & Privacy. On the General Tab click the little lock in the lower left corner to unlock the general preference pane. Then select "Anywhere" to allow Datavyu's installtion.

1. Download the latest Datavyu release for Mac OS X from the Datavyu website's Downloads Page.

2. You can drag this to your Applications folder, or some other preferred location on your computer.

Windows

1. Download the latest Datavyu release for Windows from the Datavyu website's Downloads Page. Unless you've told Windows what to do with these types of files, a *File Download* window will pop up, prompting you to either Open the Datavyu-Windows-latest.zip file, or Save it. Either option is fine - you'll merely need to navigate to the folder after downloading if you choose to *Save* rather than choosing *Open*.

2. Unzip the program by opening the datavyu folder and double-clicking on datavyu. A "Compressed (zipped) Folders Warning" will appear. Select *Extract all* to decompress the files.

3. Windows will run the Extraction Wizard. Follow the prompts and extract the files. The Datavyu program is now available, but you need to install a video plugin before you can start using video with Datavyu.
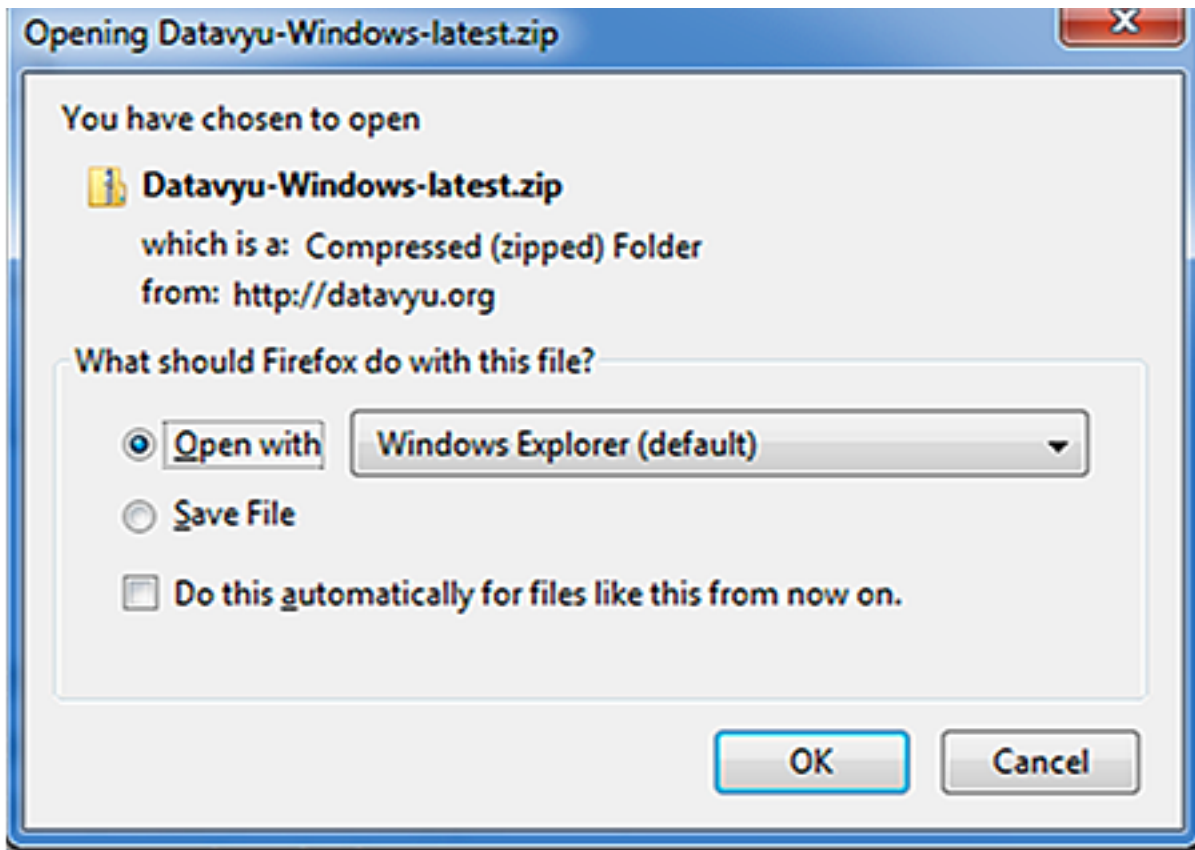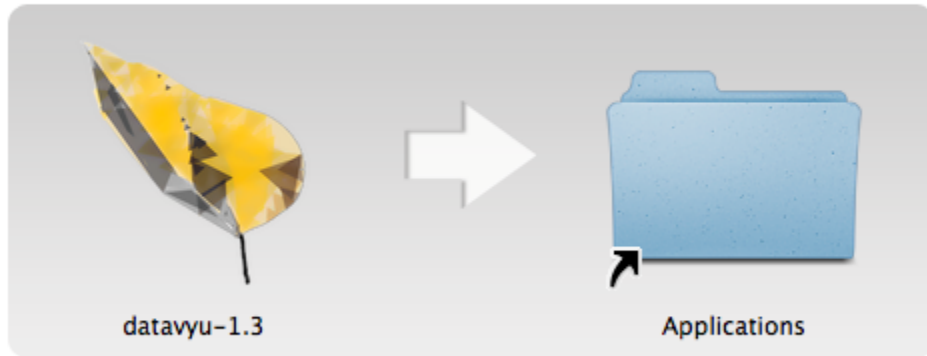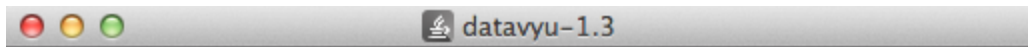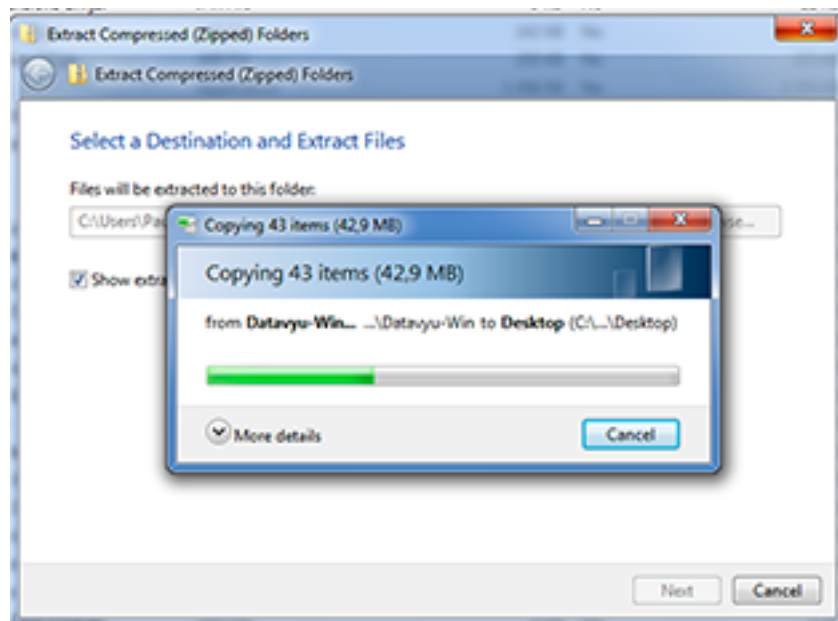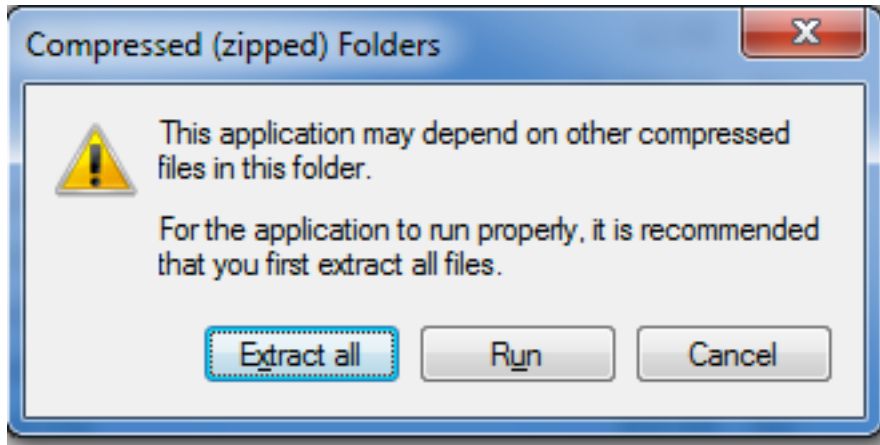
4. Install a video plugin. At present, Datavyu for Windows does not include any video plugins. To use Datavyu, you **must** download Quicktime for Windows. If you already use Apple's iTunes, you likely already have Quicktime for Windows. Otherwise, you may download Quicktime for Windows from Apple's website.

1.1.3 Keep Datavyu Up-to-Date

Every time you start Datavyu while connected to the Internet, it checks to see if your version is the latest available. When a new version is released, Datavyu will prompt you to download the latest version.

You can also check if your Datavyu version is up to date by opening the *Help* menu from the menu bar when the spreadsheet window is selected, and then selecting *Check for updates...*. If a new version is available, it will direct you to the download page where you can download the latest release. Remember to replace your current application with the latest version of Datavyu that you have just downloaded!

Important: If you have found a workflow that works well for you, please do not update your Datavyu version. We suggest coding an entire study using the same Datavyu version.

During the course of development, the Datavyu team releases several *pre-release* versions of the software prior to releasing official stable releases. Pre-releases may contain new features not yet incorporated into the main software, but are also more likely to contain bugs and to behave in unexpected ways.

To be notified of pre-release updates, simply check the "pre-release" checkbox in the updates window. Due to the increased potential for data loss or bug-related issues, you should only choose to use pre-releases if you need an unreleased feature, or are grappling with a bug in the existing stable release that is fixed in the unstable release.

Ultimately, all new features in the pre-releases are brought together and released as a new stable release.

Now that you have installed Datavyu, you can move on to Datavyu's *Getting Started Guide*.

1.2 Getting Started

Datavyu is a powerful tool that enables behavioral researchers to code observations from their video for analysis. Designed by researchers for researchers, Datavyu provides an intuitive interface for working with data sources and recording observations, and includes *an API* for more advanced data manipulation.

1.2.1 The Datavyu Interface

When you first open Datavyu, Datavyu will check if you have the latest version. If there has been a new release, Datavyu will prompt you to *update your version*.

When you launch Datavyu, you will see two windows: the *Controller*, and a blank spreadsheet. If you add a data source, a third window containing the data source will appear.

The following sections describe each component of the Datavyu interface.

Media Player

The *Media Player* is the window that presents the data source that the user is working with.

Adding a video or other data source to Datavyu is as easy as clicking on the *Add Data...* button in the Controller. For a more detailed overview of adding data sources see: *Add Data*.

Once you've added a data source, it's time to add columns, create observations, and write scripts. The *tutorials* are there to guide you.

Controller

The *Controller* allows users to control the playback of their data source and create observations in the spreadsheet. Quite literally, it controls the *Media Player*, the window that contains the video data.
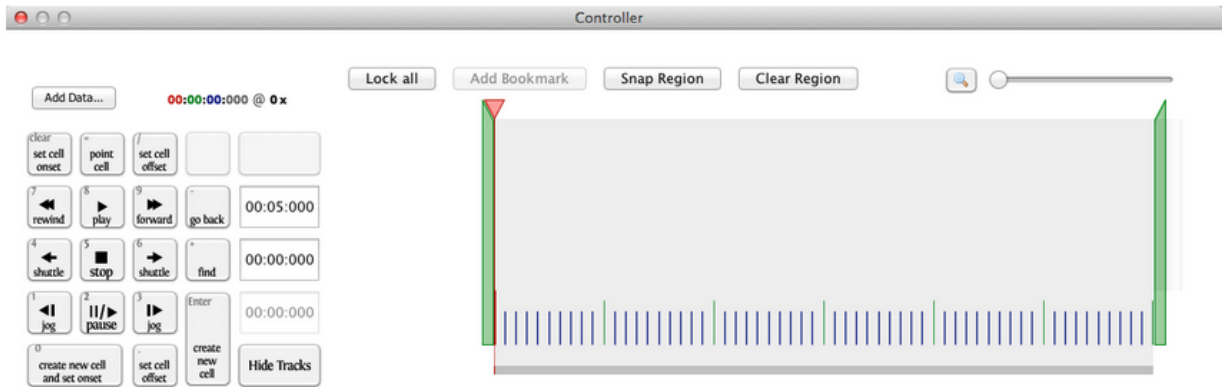
The keypad can be found on the left side of the controller. This section maps directly to the number pad on your keyboard, or to an external number pad if your keyboard does not have one and you've connected one to your computer. These keys control playback, and also enable users to set the *cell onset* and *cell offset*, create new cells and navigate within the data source. The *Add Data...* button enables you to *add a data source*.

For more about the Controller, and a detailed description of its functionality and features, see *Controller Overview*.

Spreadsheet

The *Datavyu spreadsheet* is where users record *observations* from the data source.

Figure1: This spreadsheet has four columns: "MomSpeech", "InfantSpeech", "MomObject", "BabyObject". Each column has numerous coded cells.

Being able to *Configure Columns and Codes* enables coders to record observations and link them directly to timestamps in the data source. Each user-defined column is represented by a column in the spreadsheet, and observations within a column are sorted in chronological sequence. Looking horizontally across the spreadsheet columns shows what was happening at a given point in time in the data source. Looking vertically down a column shows the sequence in which the observations occurred.

The *Spreadsheet Overview* document describes the *spreadsheet* in more detail.

Controller Overview The Controller enables users to manipulate the playback of their data source, and create new cells as they record their observations.
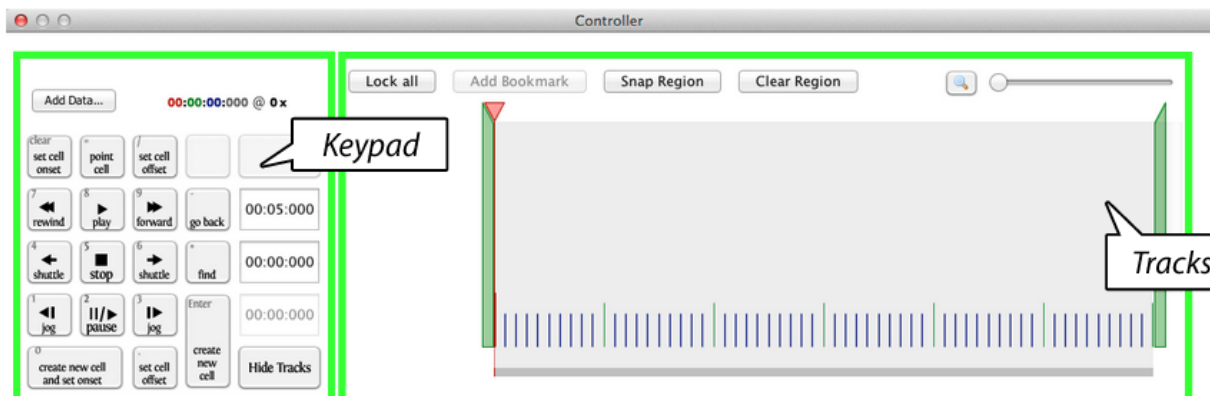
The Controller has two main areas: the keypad on the left, which maps directly to your keyboard, and the *Timeslider* on the right, which represents the current playback position.

The following sections describe the two areas in detail, and provide a useful reference for working with the Controller.

Keypad

Important: You cannot use the number keypad for inputting numbers into Datavyu. It will only perform the actions described here.

The Controller's keypad maps directly to the number pad on your keyboard or external keypad and the on-screen representation helps users remember what key performs what function. When you press a key on the keypad, a visual indicator onscreen mirrors the keystrokes. Some keypads may change the ordering of certain keys: for instance, some keypads may have a Num Lock key rather than a clear button but the Controller takes different keyboards into account and simplifies the keys as much as possible. Familiarize yourself with both the Datavyu controls and your keypad to maximize coding efficiency.

Playback Controls

- *jog left* (1): moves the playhead back one frame. If the *frame rate* is not set, *jog* will move the playhead in one second increments. Holding down *jog left* plays backward slowly.

- *pause* (2): pauses playback. Pressing *pause* again resumes playback. *Pause* only works if *play* (8) has first been pressed.

- *jog right* (3): moves the playhead forward one frame. If the *frame rate* is not set, *jog* will move the playhead in one second increments. Holding down *jog right* plays the source forward slowly.

- *shuttle left* (4): rewinds, initially at 1/32 of playback speed. Repeatedly pressing the *shuttle left* key increases the rewind speed to a maximum of 32 times playback speed.

- *stop* (5): stops playback.

- *shuttle right* (6): fast forwards initially at 1/32 of playback speed. Repeatedly pressing the *shuttle right* key increases the fast forward speed to a maximum of 32 times playback speed.

- *play* (8): starts playback.

In addition, shift *find* (+): jumps to the time of cell *offset*.

Note: For users of Mac OS X: recent versions of OS X do not allow you to hold down a key as a default. Instead it brings up a mini-menu to help you select common accents for that letter. If you want to be able to *jog* by holding down the 1 or 3 keys, you will need to open your Terminal and run the following command:

```
defaults write -g ApplePressAndHoldEnabled -bool false
```

To undo the change, run the same command but with -bool true rather than -bool false.

Coding Controls

- *set cell onset* (7): sets the *onset* for the current cell.

- *point cell* (=): creates a new cell whose *onset* and *offset* values are the playhead's current position.

- *set cell offset* (9 and .): sets the *offset* for the current cell.

- *find* (+): moves the playhead to the onset time of the current cell, which is shown in the box to the bottom right of the *find* button. Selecting a different cell will update this time.

- *go back* (-): moves the playhead back by the amount of time set in the *Jump back by* box to the right of the *go back* button. You can change the increment by selecting that box and editing the value.

- *new cell and set previous offset* (0): creates a new cell and sets its *onset* to the playhead's current time. If the previous cell does not have an *offset*, adding a new cell with this key sets the previous cell's *offset* to the current playhead time, minus one millisecond.

- *new cell and set current onset* (Enter): creates a new cell and sets its *onset* to the playhead's current time.
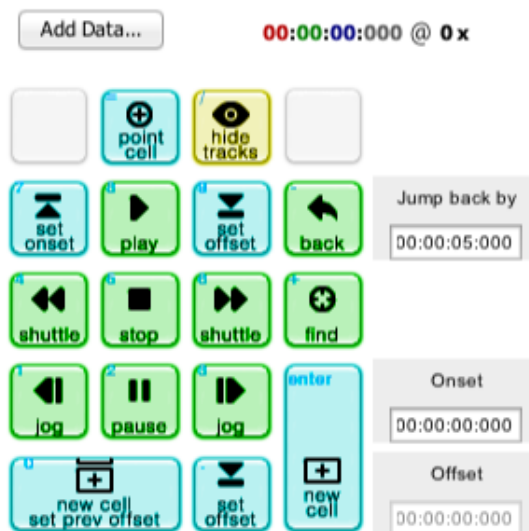
Figure2: The keys highlighted in green are *playback controls*. The keys highlighted in blue are *coding-controls*.

Tracks

Note: The tracks walkthrough video is a good way to familiarize yourself with this area of the Controller:

The *Tracks* area of the Controller helps you visualize where you are within a data source's playback. You can control the scale using the slider bar in the top right: moving it to the right zooms in, allowing you to manipulate smaller time periods than would otherwise be feasible. You can also zoom in on a selected region by clicking on the *magnifying glass* icon next to the slider bar. The red *playhead* shows the current playback position.

You can choose to focus on a single region by moving the green region boundaries. Datavyu will start playback from the leftmost boundary's position, and stop at the rightmost boundary.

The *Tracks* area of the Controller also includes five buttons:

- *Lock All*: locks the tracks to prevent the user from changing the synchronization between multiple tracks.

- *Add Bookmark*: adds a bookmark at the playhead.
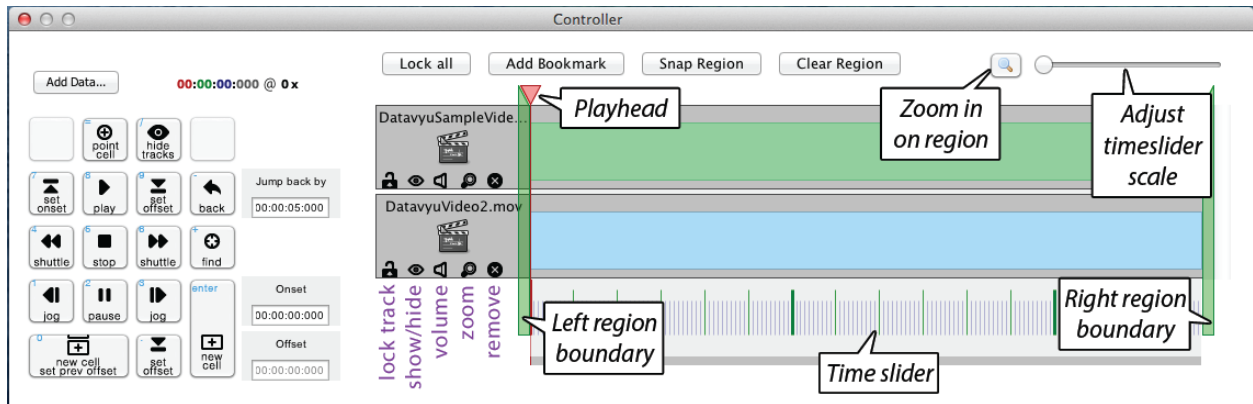
- *Snap Region*: sets the *region* to the length of the selected cell: the left boundary is placed at the cell's *onset*, and the right boundary at the cell's *offset*. You can also use the ctrl + keyboard shortcut to snap the region.

- *Clear Region*: removes the region boundaries. You can also use the ctrl - keyboard shortcut to clear the region.

- *Magnifying Glass* icon: zooms in on the timeslider to focus exclusively on the snapped region.

Add Data

Important: Do not use the VLC plugin because it is not fully implemented and tested. It has frame accuracy problems and therefore is not suggested. Please use Quicktime instead. If your videos do not play smoothly in Quicktime, please convert your videos if you would still like to use Datavyu.

Datavyu currently supports video through QT. To convert video file types, see: *Convert File Formats*. If you are going to be working with multiple data sources see: *Code Multiple Data Sources at Once.*

Providing you have adequate processing power and a sufficiently strong graphics card, you should be able to work with high-resolution video files without challenge.

To add a data source:

1. Click the *Add Data...* button in the top left corner of the Controller.

2. A file selection window will open. Select the data file you will be coding.

3. If the data source is a video file and Datavyu is unable to determine its *frame rate*, it will ask you what the video's frame rate is. You can determine the frame rate by opening the video in QuickTime Player, and selecting *Show Movie Inspector* from the *Window* menu. The Inspector presents information about the video. The frame rate is labeled *FPS*, or frames-per-second.

   Datavyu uses the frame rate to accurately play and *jog* through videos, so it is important that you set the correct frame rate.

4. If Datavyu cannot read your video's frame rate or if it reads the wrong frame rate. You can manually set the frame rate by double clicking on *Steps per second* and writing the correct frame rate. Please press *Enter* and Datavyu will use your new frame rate.

   See also:

- *Spreadsheet Overview*

- *Tutorials*

Spreadsheet Overview   The Datavyu Spreadsheet is where coders record *observations*. The spreadsheet is the core of Datavyu. Coders can record observations and link them directly to timestamps in the data source when they *Configure Columns and Codes*.

For a brief introduction to some of the spreadsheet's components and capabilities, watch the spreadsheet walkthrough video:

Each user-defined column has its own column in the spreadsheet. *Cells* are column entries, boxes in the column, where coders record their observation data as *codes*. Cells follow each other in sequence: looking vertically down a column shows the sequence of observations for that column.

Datavyu automatically links the times coded in the spreadsheet to the current time in the data source. This allows coders to record the *onset* and *offset* times of events in the spreadsheet. Coders can also jump to a relevant time in the data source by selecting a specific cell in the spreadsheet and pressing find (+) on the Controller.

Spreadsheet Tabs   Datavyu allows users to open multiple spreadsheets at one time. When the program opens, it opens a blank spreadsheet. This spreadsheet can be used to create a new file or it can be closed if you are working on pre-coded spreadsheets. Feel free to open as many spreadsheets as desired.

Users can work on multiple spreadsheets at one time. When finished with coding, please be sure to save each individual spreadsheet.

Columns   Datavyu uses columns to group together related observations. In general, coders will code the data source column-by-column meaning that they code one entire column before coding a new column.

Using the Code Editor, you can configure columns to represent any number of observations.

Columns have *codes*, which represent the feature that you are observing. For instance, a code could be "Left hand touch", or "Smiling", or "Look left." When coding the data source, coders can record the presence or absence of these codes and/or potential values within them. Columns can have as many or as few codes as you want. If you want to score durations without scoring codes, you can leave the default code as is and ignore the <code01> prompt.

Also note that column names are limited to the letters of the alphabet, numbers and the underscore symbol (but numbers and underscores cannot be the first character of the name) to eliminate potential confusion in scripting and SPSS analyses. The *column configuration* tutorial provides instructions for configuring columns.

Figure3: A typical Datavyu spreadsheet with an example of five columns and Temporal Alignment turned on. Note the *plus* icon in the top right, which you use to add new *columns*.

Hide and Show Columns To hide a column, select the column by clicking on its name at the top of the spreadsheet. The selected column will have a blue background. You can select multiple columns by Cmd -clicking (on Mac) or Ctrl -clicking (on Windows). Then, in the *Spreadsheet* menu, select *Hide Selected Columns*.

You can also control each column's visibility from the *Column List*, which you access from the *Spreadsheet* menu. The *Column List* shows all of the columns in the current spreadsheet, and includes a checkbox, which you can use to toggle column visibility.

If you wish to show all columns you can do so with the *Show All Columns* menu option from the *Spreadsheet* menu.

Rearrange Columns To rearrange the order of columns within the spreadsheet, simply click the column's name and drag it left or right to the desired location.

Cells Each cell represents an observation scored by the coder.
Minimally, Datavyu displays three values for each cell, but you can ignore one or all of these values.

- *onset*: the first time value displayed on the top of the cell. If you don't code this value, the cell will display the default value of 0:00:00:000. You can code a time value to mark the beginning of an event or to tag the approximate time of an event.

- *offset*: the time value displayed on the top right of the cell. If you don't code this value, the cell will display the default value of 0:00:00:000. You can code a time value to mark the end of an event or to tag the approximate time of an event.

- *ordinal*: the cell ordinal indicates the position of the cell within the column. The first cell (the one with the earliest onset or the first cell you code if you do not mark onsets) would be number 1, the second number 2, and so on. Ordinals are automatically coded and updated as you code your data source. You will never need to set the ordinal.

The following image labels each component within a Datavyu cell.
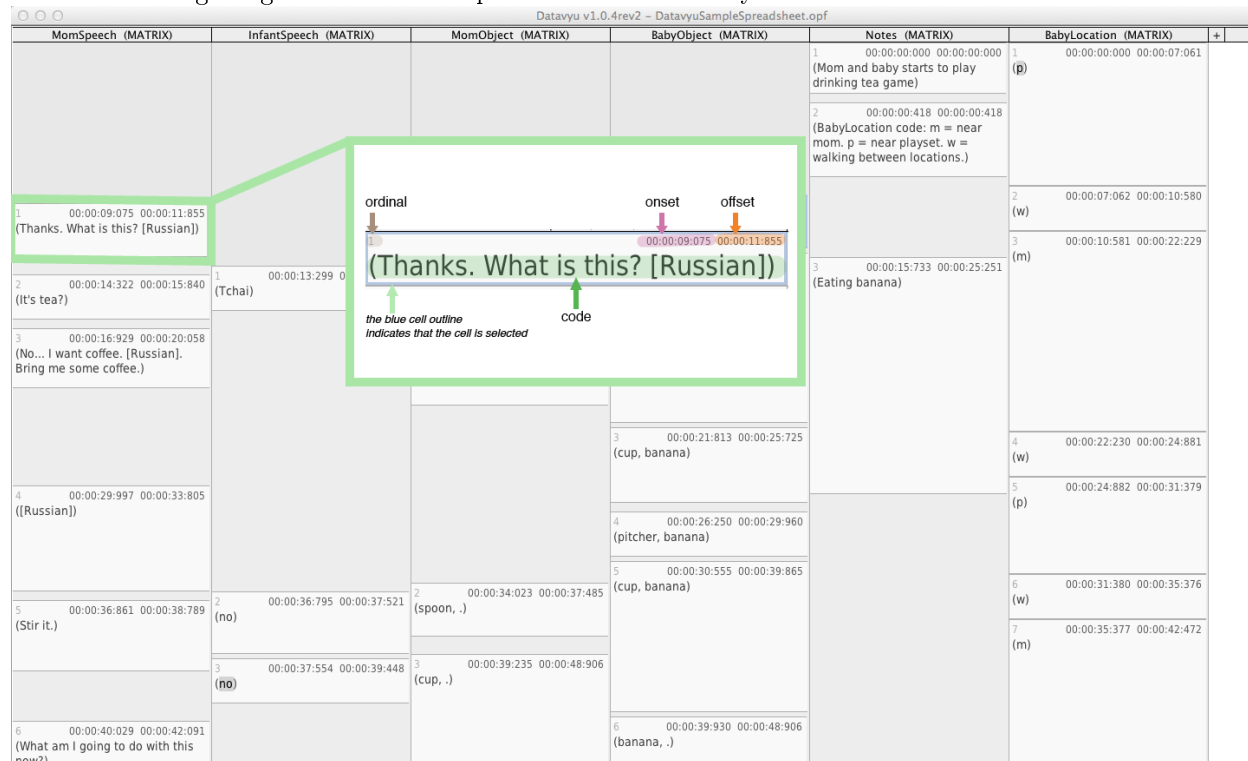
Spreadsheet Customization   Datavyu allows you to modify the spreadsheet by including options that you can activate or deactivate, depending on your needs.

Zoom   By default, the Datavyu spreadsheet uses one font size for the user-input codes in cells. You can increase this font size by choosing *Zoom* from the *Spreadsheet* menu. However, the font size for the onset and offset times do not change.

You can also modify zoom using keyboard shortcuts:

- cmd + / alt + zooms in

- cmd - / alt - zooms out

- cmd 0 / alt 0 resets the zoom to the default level

Temporal Alignment   Temporal alignment is an important feature of Datavyu. When temporal alignment is active, Datavyu groups cells based on their onsets and offsets, visually representing the period of time each cell occupies. This Temporal Alignment video highlights the differences between active and inactive temporal alignment:

Temporal alignment allows you to visualize what occurred at what time so you can compare event sequences across columns. When coding, you should ensure that temporal alignment is active.
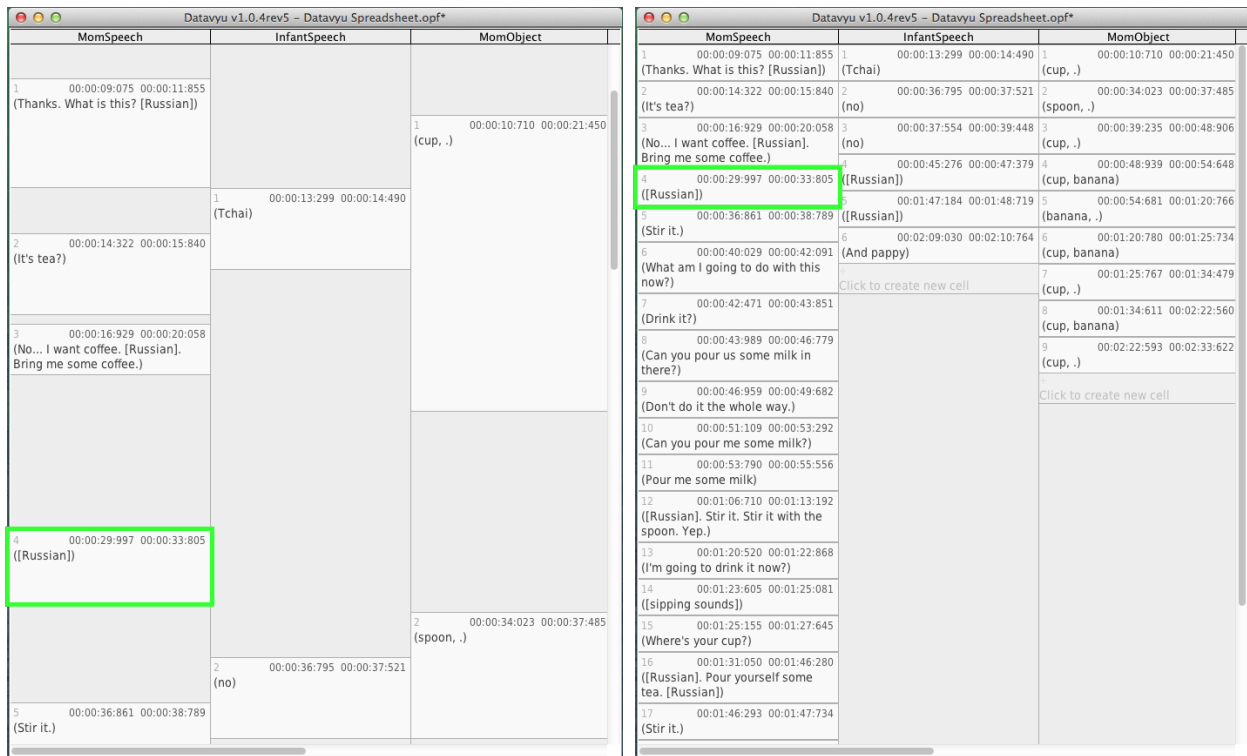
Figure4: At left is a spreadsheet with temporal alignment enabled; at right is the same spreadsheet with temporal alignment disabled. Note that the cells contain the same information, it is only their presentation changes. Note also that enabling temporal alignment makes it easier to code and alllows you to visualize the relative length of events and their relations across columns.

Toggle Temporal Alignment   You can toggle temporal alignment using the cmd T keyboard shortcut, or by selecting *Temporal Alignment* from the *Spreadsheet* menu.

16

Keyboard Shortcuts   We have provided a simple list of the major shortcuts used in the Datavyu Spreadsheet.

Action	Mac	PC
Temporal Alignment	⌘ T	Ctrl T
Snap to region	Ctrl +	Ctrl +
Clear Snap Region	Ctrl -	Ctrl -
Jump to Current Onset	+	+
Jump to Current Offset	Shift +	Shift +
New Cell to the Left	⌘ L	Ctrl L
New Cell to the Right	⌘ R	Ctrl R
New File	⌘ N	Ctrl N
Open	⌘O	Ctrl O
Save	⌘ S	Ctrl S
Save as	⌘, S, ⇧	Ctrl, Shift, S
Quit Datavyu	⌘ Q	
Hide Datavyu	⌘ H	
Hide Other Windows	⌘, –, H	
Code editor: add column	⌘ M	Ctrl M
Code editor: add code	⌘ A	Ctrl A

Action	Mac	PC
Temporal Alignment	CMD T	CTRL T
Snap to Region	CTRL NUM+	CTRL NUM+
Clear Snap to Region	CTRL NUM-	CTRL NUM-
Jump to Current Onset	NUM+	NUM+
Jump to Current Offset	SHIFT NUM+	SHIFT NUM+
New Cell to the Left	CMD L	CTRL L
New Cell to the Right	CMD R	CTRL R
New File	CMD N	CTRL N
Open	CMD O	CTRL O
Save	CMD S	CTRL S
Save As	CMD SHIFT S	CTRL SHIFT S
Quit Datavyu	CMD Q	
Hide Datavyu	CMD H	

1.3   Tutorials

Datavyu Tutorials provide comprehensive guidance for common tasks you might wish to perform.

1.3.1   Add a Column

Add a column to create a new coding pass or a new set of codes.

1.3.2   Rename a Column

Rename a column to better reflect your coding pass, to increase transparency, etc.

1.3.3   Delete a Column

Delete a column if the information is no longer needed.

1.3.4 Configure Datavyu Codes

Configure columns and codes to set up a spreadsheet for coding.

1.3.5 Add Cells

Add cells while coding or annotating a video file.

1.3.6 Delete a Cell

Delete a cell if the information is not needed.

1.3.7 Export Data

Export Data from Datavyu into a statistical package, into a text file, etc.

1.3.8 Use Scripts to Automate Tasks

Use scripts to automate tasks such as inserting, deleting, and modifying cells.

1.3.9 Code Multiple Data Sources

Code multiple data sources at once e.g., two or more videos recorded at the same time.

1.3.10 Convert File Formats

Convert file formats into an appropriate format for Datavyu.

Add a Column

Datavyu represents sequences of events as columns in the spreadsheet.

Adding a column to your Datavyu project is simple. The following steps will guide you through the process.

1. Open the Datavyu spreadsheet.

2. Click on the plus sign in the top right of the spreadsheet. The New Column window will open.

3. Input a name for it.

4. Select *OK*. The column will be added to the spreadsheet.

You can also add a column when you *Configure Columns and Codes* through the code editor.

Rename a Column

Datavyu provides four different ways to rename a column using the Datavyu user interface.

Rename Directly from the Spreadsheet

1. Open the Datavyu spreadsheet.

2. Double click on the column name. The *New Variable Name* window will open.

3. Type in the new name for the column, and press Enter or click *OK*.

Rename Using the Spreadsheet Menu   You can also change a column's name from the *Spreadsheet* menu.

1. Open the Datavyu spreadsheet.

2. Select the variable name you wish to change. Selected variables have a blue background.

3. Select the *Spreadsheet* menu, and then *Change Variable Name*. The *New Variable Name* window will open.

4. Type in the new name for the column, and press Enter or click on *OK*.

Rename From the Column List   Using the *Column List* is another way you can change a column's name. The *Column List* also has the advantage of allowing you to change multiple column names in one place.

1. Open the Datavyu spreadsheet.

2. Open the *Spreadsheet* menu, and then select *Column List*.

3. Double click on the name of the column whose name you want to change, type in a new column name, and press Enter.

Rename Columns Using the Code Editor   Datavyu's *Code Editor* enables you to configure *columns' codes*, but you can also edit the names of your columns' and codes'.

1. Open the Datavyu spreadsheet.

2. Open the *Spreadsheet* menu, and then select *Code Editor*.

3. Double click on the name of the column whose name you want to change, and type a new column name. The spreadsheet will update automatically.

   See also:

   - *Spreadsheet Tabs*
   - *Add a Column*
   - *Delete a Column*

Delete a Column

To delete a column,

1. Open the Datavyu spreadsheet.

2. Select the column you wish to delete. The background will turn blue when selected.

3. Open the *Spreadsheet* menu, and select *Delete Column*.

Configure Columns and Codes

Each column may have a collection of one or more *codes*. Each code has a value that the coder inputs while coding a data source.

   Datavyu provides a Code Editor for configuring columns. Check out our video walkthrough of the Code Editor's features:

Open the Code Editor From the spreadsheet, click on the *Spreadsheet* menu, and then select *Code Editor*.
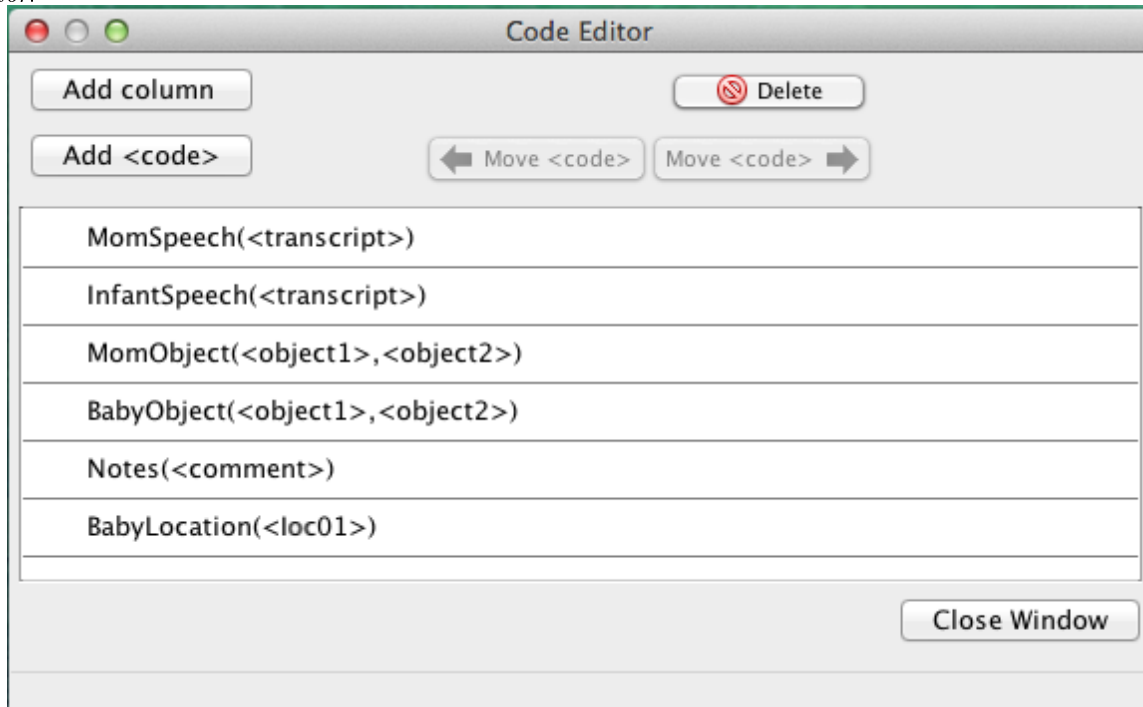
The Code Editor window will open and list all existing columns and codes. From here, you can add and edit columns and codes.

Add a Column Click on the *Add Column* button. A new column called "column1" will appear in the list of columns. By default, "column1" has one code, "<code01>".

You can change the name of the column or code by double-clicking on its name and typing a new name.

Add Codes To add a code to a column, select the column. You'll know you have selected it when its background is light blue. Then, click on the *Add <code>* button.

You can change the name of the code by double-clicking on its name and typing a new name. To reorder a column's codes, select the code by clicking on it, and use the *Move <code>* buttons to move it.

Rename Codes To rename a code, open the Code Editor, and double click on the name of the code you want to change, and type the new name. The spreadsheet will update automatically.

Remove Codes To delete a code, select it by clicking on it, and then click on the *Delete* button in the top right.

Video Example ──
Note: The next video displays how to use the code editor to set up a spreadsheet from scratch.
──

Add Cells

Datavyu represents events as *cells* in the Datavyu spreadsheet. Cells relate to *columns* because they capture the events that you are coding in that pass.

This tutorial assumes that you have already configured your spreadsheet to include a column. If you have not already created a column in your spreadsheet, start with the *Add a Column* tutorial.

There are three ways to add cells to a Datavyu spreadsheet: using the spreadsheet menu, using *Click to Create New Cell* in the body of the spreadsheet, or using the Controller keypad. Controller keys are ideal for creating cells while coding a video. Note that Controller keys give you more options for how to insert cells and onset/offset times.

Tip

Use the tab and shift-tab keyboard shortcuts or the arrow keys to easily navigate among codes within a cell.
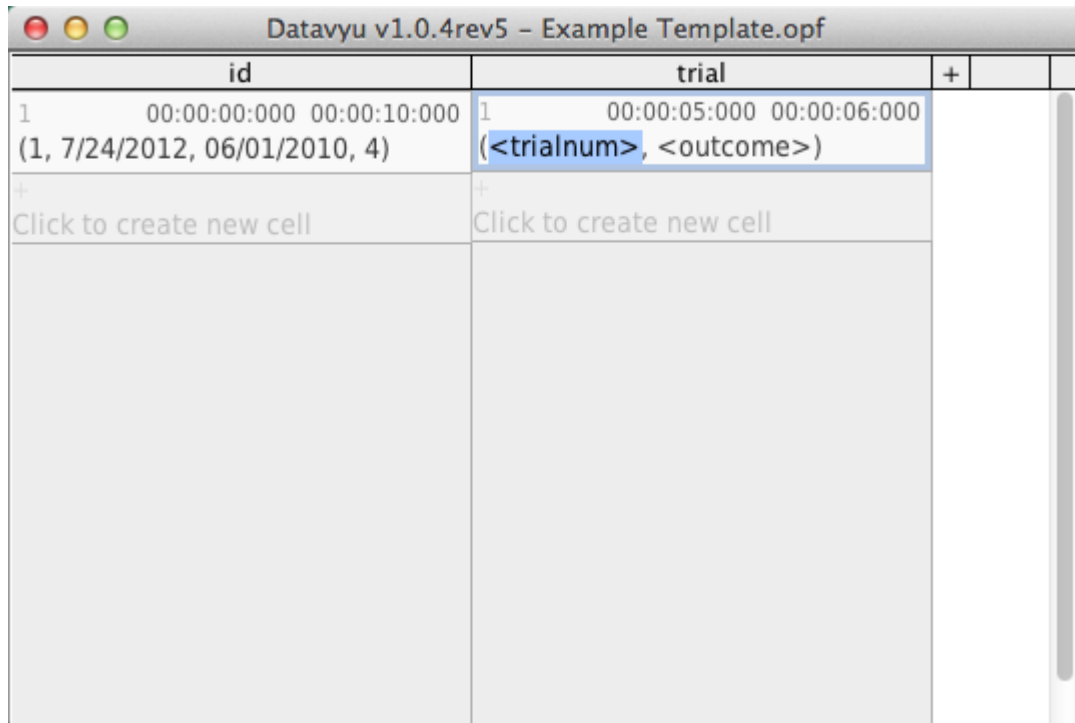
Add Cells Using the Spreadsheet Menu Using the spreadsheet menu to add cells to an existing column is simple:

1. Click on the *Spreadsheet* menu and select *New Cell*. A new cell appears with the current play head time as its onset and placeholders for any codes configured for the column.

2. Set the cell offset using the Controller.

You can also add cells to neighboring columns by selecting *New Cell to Left* or *New Cell to Right*.

Add Cells Directly in the Body of the Spreadsheet Adding cells to an existing column is simple:

1. Click in the gray *Click To Create New Cell* area. A new cell appears with the current playhead time as its *onset* and placeholders for any codes configured for the column.

2. Record appropriate values for the codes. For the "trial" column in the example spreadsheet, that means filling out the <trialnum> and <outcome> codes.

3. Set the cell *offset* using the keypad on the Controller.

Add Cells from the Controller   Adding cells using the controller is even easier than doing so from the spreadsheet:

1. Navigate to the point in the data source that you want to be the cell's *onset*. Use the *new cell and set previous offset (``0``)* key on the keypad to create a new cell, set its onset based on your location in the data source, and set the previous cell's onffset to 1 ms prior all in one go.

   or

   Create a new cell using the *new cell* (Enter) key on the keypad. The onset will reflect your location in the data source.

**Delete a Cell**

To delete a cell:

1. Open the Datavyu spreadsheet.

2. Select the cell you wish to delete. The cell will be outlined in blue when selected.

3. Open the *Spreadsheet* menu, and select *Delete Cell*.

Export Data from Datavyu

Datavyu provides an integrated export tool for exporting Datavyu data. To export your data, select the *File* menu, and then select *Export File*. This outputs data to a CSV file that has one column for every code in the spreadsheet: *ordinal*, *onset*, and *offset*, as well as user-configured codes.

If this format does not work for the analyses you need to perform, Datavyu supports Ruby scripting, which you can use to create a script that exports the data in your desired format.

For a detailed guide to exporting data using Ruby scripts, see: *Use Scripts to Export Data from Datavyu* in the Datavyu Ruby API documentation.

See also:
Use Scripts to Automate Tasks for instructions on running scripts within Datavyu.

Use Scripts to Automate Tasks

Datavyu provides a full suite Ruby scripting API to help you focus more time on coding and spend less time performing routine tasks.

The *Ruby API documentation* guides you through writing scripts and provides context to help you become an adept Ruby script-writer regardless of your programming experience. You can also watch a video version of this tutorial:

Run Scripts   Before you can run a script, you must write one. Refer to the *Ruby API documentation* for scripting help.

1. Save your script as a .rb file, and put it somewhere you will be able to find again, such as a Scripts folder on your desktop, or the location where you store your Datavyu files.

2. In Datavyu, select the *Script* menu. The *Script* menu has two options: *Run script* and *Run recent script*.

3. Select *Run Script* to choose a script that you have saved.

4. A file selection window will open. Navigate to the correct folder and choose the script you wish to run.

5. The Datavyu **Scripting Console** will open and run your selected script and display any errors that may arise. If Datavyu reports a script error, use the provided information to find it.

6. You can close the Scripting Console when you receive a notification that the script has run successfully.

Scripts that you have recently run will be listed in the *Run Recent Scripts* menu in the *Script* menu. This makes it easy to repeatedly run the same scripts.

Code Multiple Data Sources at Once

With Datavyu, you can code multiple data sources in one spreadsheet. For instance, if you have three different camera views of an experiment, you can bring them together into Datavyu, and code them as one.

Adding multiple data sources is easy, but aligning them perfectly can take some effort. The following sections guide you through the process.

Add Multiple Data Sources   To add multiple data sources, simply add a data source using the Data Viewer Controller's *Add Data* button. You can add as many data sources as you desire.
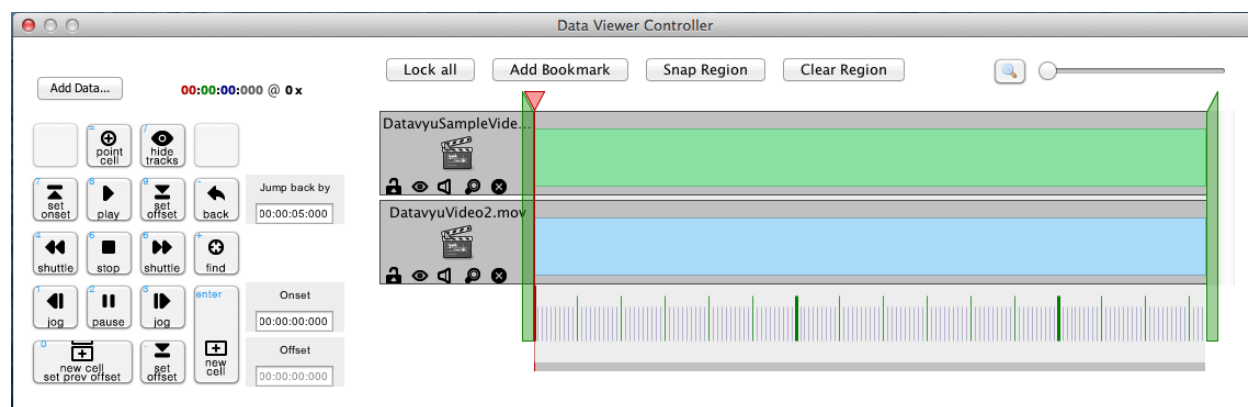
Each data source has its own line in the *tracks* area of the Controller. You can select an individual data source by clicking on its blue bar. When selected, the bar will turn green.

Align Tracks   To code your multiple data sources, you need to align them so that times recorded in the spreadsheet are accurate for all data sources.

You will need to locate an event that you can use as the basis for synchronizing your sources. For example, if the lights flashed during the experiment, you could align the tracks at that point.

Align Tracks Manually   To align your data sources, select one and drag it to align with the other source. Increasing the zoom using the *slider bar* in the Controller can give you more granular control, and facilitate precise alignment.

When you have successfully aligned your tracks, select *Lock All*. This prevents you from accidentally desynchronizing the tracks. You can also lock a single track by clicking on its *lock* icon to the left of the time slider. This can be helpful while synchronizing multiple tracks.

Align Tracks Using Bookmarks   You can also use bookmarks to help synchronize tracks. If you identify a distinct synchronization point, navigate to that point in the first track, and click the *Add Bookmark* button. This will create a bookmark in that track.

Then, locate the same event in the other data source and bookmark it. When you drag the tracks to align them, they will "snap" and align both bookmarks.

You can repeat this process with all the tracks that you need to synchronize, and then select the *Lock All* button to lock their arrangement.

Saving Between Datavyu Uses   When working with multiple data sources, Datavyu saves the data source synchronization and bookmarks when you save the spreadsheet. This way, when you reopen the spreadsheet to do more coding, or to work with the data, you will not need to re-synchronize the data sources. This also helps ensure consistency between coding passes and reliability coders.

Convert File Formats

Datavyu currently supports video through QT and VLC.

If your data source is not directly supported by Datavyu, you may be able to convert it to a readable format using Datavyu's Convert Videos tool.

1. From the Spreadsheet, select the *Controller* menu, and then choose *Convert Videos*.

2. Select a source video. This will be the video you are converting.

3. Select a target video. This is the video that Datavyu will output or convert your source video into.

4. Select *Convert Video*, and your video will be converted.

For help with data source conversion or other import issues, see Datavyu's Support site.

1.4   Reference

1.4.1   Glossary

API

An API (Application Programming Interface) enables disparate software components to interact with each other by specifying functions or routines to perform tasks. Datavyu's API uses the Ruby programming language.

argument

The information the user specifies to a method for a given parameter. For instance, "charlie" might be the argument specified for the name parameter. See: *Classes, Methods, and Parameters*.

cell

A cell is a graphical representation of an observation. Cells are rectangles that stack in the columns of Datavyu's spreadsheet and contain the observation data that a coder inputs when coding a data source.

class

A pre-defined object type, that has associated attributes and methods. See: *Classes, Methods, and Parameters*.

class method

A *method* that belongs to a *class* and must be invoked on an instance of that *class*. See: *Classes, Methods, and Parameters*.

code

Datavyu codes are column components that researchers are observing. Each Datavyu *column* may have multiple codes for which a coder will record observations. For instance, a column called "step", which refers to walking, might have a "foot" code that would differentiate between left or right feet, and a "direction" code that indicated if the person was stepping forward or backward.

coding manual

Documentation for codes in a Datavyu spreadsheet. Should be written as if instructing someone with no prior information about the spreadsheet or project.

coding pass

A pass intended to fill in all fillable codes in a column. Coding passes can be aided by previous passes so that work is not repeated.

column

Datavyu columns are phenomena or events that observations are collected about. Columns are *key-value pairs* that associate a column name with a variety of *codes*. Columns can be general, such as "trial" or "ID", or can be specific, such as "step" or "hand". Datavyu represents columns as a column in the *Spreadsheet*.

comment

A code that consists of freeform text. Useful for recording unique observations in the data.

Controller

The Controller is a core Datavyu window that enables you to control playback of data sources, and to record observations. See: *Controller Overview*

data

The audio or video files that are being studied.

frame rate

Frame rate (also known as frame frequency) is the frequency (rate) at which an imaging device produces unique consecutive images called frames. The three main frame rate standards are: 24p, 25p, and 30p.

integer

A whole number. Integers are numbers that are not fractions, and thus, which have no numbers after a decimal point.

key-value pair

A data representation that pairs "keys" (some thing you have data about) with "values" (the data itself). A key-value pair could be "telephone" and "555-123-4567", for instance. Sometimes, values are represented as a list of values, or as an array.

method

A defined action that you can perform. Methods may accept arguments in order to perform their specified tasks. See: *Classes, Methods, and Parameters*.

observation

An instance of the *column* that is being coded.

offset
cell offset

The end time of a cell.

onset
cell onset

The start time of a cell.

ordinal

The position of a *cell* within its *column*. The ordinal is indicated in the top left corner of each cell in Datavyu.

parameter

The information that is methods use to perform their tasks. Parameters can be required or optional. See: *Classes, Methods, and Parameters*.

playhead

The playhead indicates the current point of playback in the *timeslider* of the Controller.

region

The area of the *timeslider* that has been brought into focus using the timeslider's brackets.

reliability column

A Datavyu column that is a copy of an original column used to ensure that the coded observations are accurate.

script

A program that performs a specific task.

spreadsheet

The core Datavyu window where coders record observations. See: *Spreadsheet Overview*

standalone method

A *method* that does not belong to a specific *class*. See: *Classes, Methods, and Parameters*.

string

In computing, a string is a linear sequence of characters. Strings can be random characters, words, or sentences.

timeslider

The timeslider represents the length of the data source, and enables users to visualize where the *playhead* is with respect to the entirety of the data source.

1.4.2   Naming Restrictions

Datavyu places certain restrictions on code and column names:

- Column names **should not include periods**.

- Column names must begin with a letter.

- The underscore is the **only** permitted special character.

- Datavyu also restricts certain names, making them unavailable for use. The following names cannot be used as a column or code name:

    -several.words.separated.by.periods -1stCharacterIsANumber -i_really_really_mean_it!!!!!!

If you're new to coding, or want to bone up on advice from the experts, consider the *Best Practices Guide* for instruction on how best to harness Datavyu's capabilities, and tips for starting to code behavioral videos.
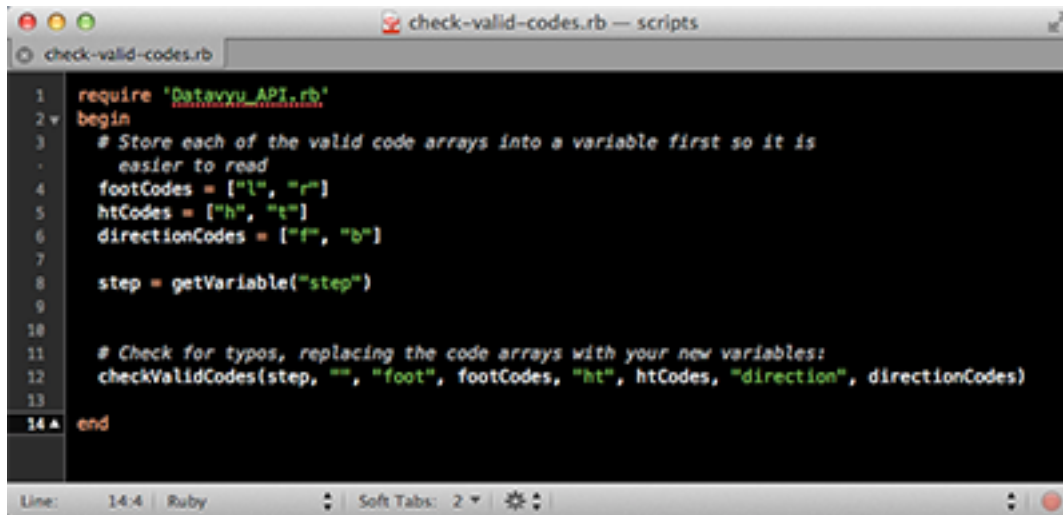
Chapter 2

Ruby API

The Datavyu scripting API provides a scriptable interface to Datavyu's spreadsheet, allowing you to manipulate your data, export in any format you'd like, or check your data for errors.

2.1 Getting Started with APIs and Scripting

APIs, Application Program Interfaces, are collections of code that specify how software components communicate with each other. Datavyu's API uses Ruby programming language, a popular object-oriented language. Ruby is reasonably easy to learn, and novice programmers can start writing scripts with minimal programming knowledge.

Before you dive into the *API Tutorials*, though, you should familiarize yourself with *what an API is*, learn about Ruby's *classes* and *methods* and how they differ, and follow the *Introduction to Scripting*.

2.1.1 Introduction to APIs

APIs, Application Program Interfaces, are collections of code that allow software components to communicate with each other.

The Datavyu API lets users write *scripts* in a Ruby programming interface that can modify the contents of Datavyu spreadsheets.

Suppose you have a dataset that contains fifty Datavyu files and videos. After you analyze your data, you realize that you want to go back and code a new pass for all of your files. Instead of manually creating a new column in all fifty Datavyu files, you can write a script that will do it for you! The script can include any and all desired codes within the column, and will save time and reduce the potential risk of error.

Or suppose you want to change the name of a code. You can use a script to make the changes across all the relevant files.

Or perhaps you'd like a reliability coder to score 25% of each participant's video. You can use a script to create a "reliability column" and insert every fourth cell (or a randomly selected 25% or any other metric) for a specific column.

Or maybe you want to check each file for typos or impossible onset/offset times. Use a script for data checking!

Computers are ideally suited for doing repetitive tasks quickly and conistently. The Datavyu API enables a researcher to perform these kinds of tasks programmatically so that doing a repetitive task becomes a matter of writing a few lines of computer code and running the script.

Writing scripts with the API is straightforward and accessible to even a novice programmer. The ability to use scripts to automate tasks greatly increases the ways that Datavyu can facilitate data coding and analysis, and is well worth overcoming the initial learning curve.

Ruby and You

For a good introduction to Ruby, consider going through the Ruby in Twenty Minutes tutorial, which provides a quick overview to working in Ruby.

You do not need to be an experienced programmer or even understand the ins and outs of Ruby to be able to write scripts with the Datavyu API. Following the *API tutorials* is a good way to become familiar with scripting for Datavyu. With some practice, you will likely be able to customize the sample scripts for your own purposes. However, gaining a grounding in some programming concepts will make it easier to create your own scripts for your unique use cases and needs.

Consider *Classes, Methods, and Parameters* for an introduction to object oriented programming, *classes*, and *methods*. Also, *Introduction to Scripting* discusses the requirements you should keep in mind when you start writing scripts.

2.1.2   Classes, Methods, and Parameters

Ruby, the programming language that the Datavyu API uses, is an object- oriented language.

Classes

Object-oriented languages represent concepts, like Datavyu's columns or cells, for instance, as "objects". Objects come in different types, or "classes". For example, objects that represent numbers, strings (written text), or Datavyu columns have different classes.

An object can have a variety of attributes depending on the data that pertains to the object. In Datavyu, a cell object's attributes would be its codes; a column object's attributes might be its cells, its name, and its onset and offset times. The class of an object defines what types of attributes a specific type of object will have.

Thinking about Datavyu specifically, the "column" class describes the Datavyu columns. The "trial" object in the example spreadsheet would then be an instance of the "column" class. This is in fact how the Datavyu API works. The Datavyu API includes two classes to represent Datavyu concepts: RColumn describes Datavyu's *columns*, and RCell describes Datavyu's *cells*.

Methods and Their Parameters

While classes describe objects and their attributes, *methods* define actions that you can perform on an object. The Datavyu API defines numerous methods to help you manipulate, modify, add, and delete data in the Datavyu spreadsheets.

In order to work, many methods need additional information. For instance, getColumn(), retrieves a column from a Datavyu spreadsheet so that you can modify or update it using your script. But it needs you to tell it what column it should retrieve. Each method defines what information it can receive. These are called *parameters*. getColumn() has one parameter: name, which is the name of the column that you wish to retrieve.

Parameters are the types of information that you can specify for a method, and are specified in the method's definition (you can view all of Datavyu's method definitions on the *API reference page*).

The information you actually provide when you use the method is called an "argument". Arguments are user-specified. For getColumn(), you might want to retrieve a column called "trial", so "trial" would be your argument for the name parameter. Again, parameters are part of the method's definition, while arguments are the information you provide to the method.

For example, the add_codes_to_column() method enables users to add one or more codes to a column. add_codes_to_column() has two parameters: column, the name of the column you want to update, and *codes. The * indicates that the parameter is a list: so you can specify one (or more) codes as a list of *Strings*.

Suppose you wanted to add two codes, "leftHand" and "rightHand" to the "arm" column. "arm" would be your argument for the column parameter, and "leftHand", "rightHand" your argument for the *codes parameter. column and code are specified in the add_codes_to_column() method definition, and "arm", "leftHand", and "rightHand" are your user-specified arguments.

Method definition:

```
add_codes_to_column(column, *codes)
```

Method call:

```
add_codes_to_column("arm","leftHand","rightHand")
```

Standalone Methods and Class Methods

There are two different types of methods: standalone methods and *class methods*. Class methods act directly upon an instance of a specific class. Standalone methods perform actions on their own.

getColumn() is a standalone method that you use to retrieve a column from the Datavyu spreadsheet so that you can modify it with your script. To retrieve a column called "trial" from the spreadsheet, you would run:

```
getColumn("trial")
```

In contrast, add_code() is a class method of the RColumn class, and *is invoked* on an RColumn object. Assuming that you have already retrieved a column called "trial" from the spreadsheet, and assigned it to an RColumn object, the following code would add the *newCode* code to the "trial" RColumn object:

```
trial.add_code("newCode")
```

Comparing them side-by-side can help highlight the difference:

```
getVariable("trial")        vs.        trial.add_arg("unit")
```

Standalone Method

instance   Class Method
of class

Standalone Methods
are of the format:
```
method()
```

Class Methods
are of the format:
```
instanceName.classmethod()
```
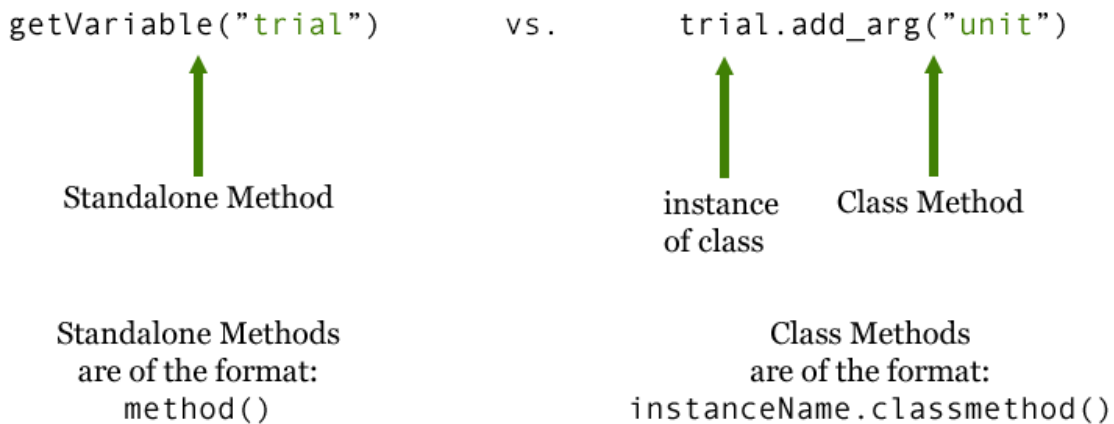
Understanding the difference between standalone and class methods will make it easier for you to easily use the methods included in the Datavyu API.

Next Steps

Now that you have a grounding in the difference between classes and methods, parameters and arguments, and know how to invoke both standalone and class methods, consider our *Introduction to Scripting* for tips

before diving into the *API Tutorials*.

2.1.3   Introduction to Scripting

This detailed guide describes how to write Ruby scripts to automate tasks in Datavyu and to ensure reliable *coding*.

Classes, Methods, and Parameters provides an overview of the *methods* and *classes* of the Datavyu API. For a more in-depth discussion of each particular method and function, refer to the *Reference* documentation.

Recommended Text Editors

Ruby scripts are simply text files with a .rb file extension. You can write scripts in any text editor, including built-in ones like Notepad or TextEdit. However, while those programs are adequate for scripting purposes, modern text editors make scripts much easier to read by providing syntax highlighting, which can make a world of difference when attempting to debug an issue.
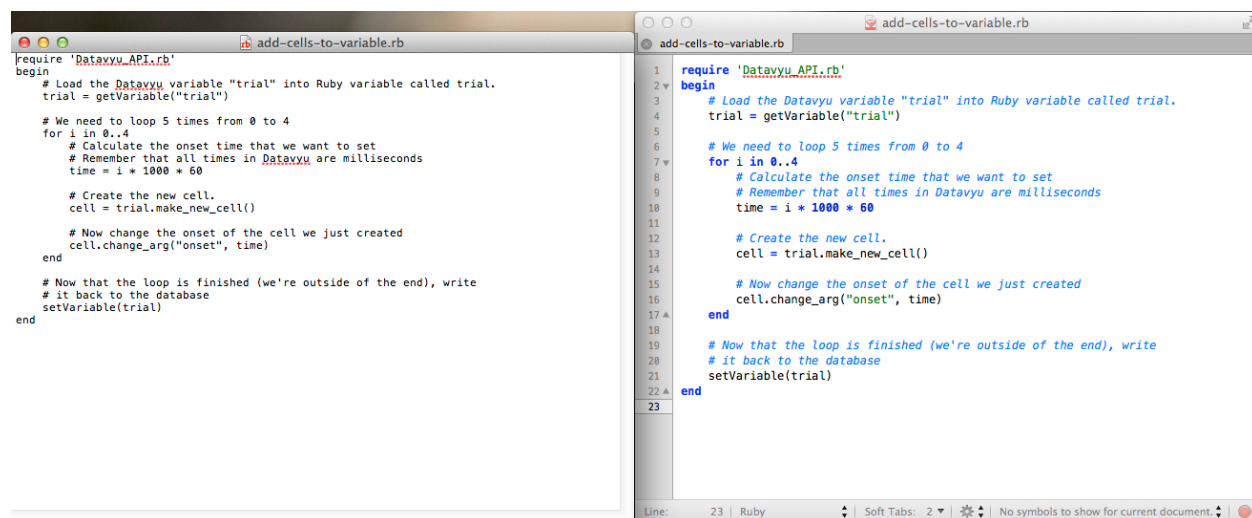
Figure1: At left: TextEdit (Mac OS X). At right: TextMate 2 (Mac OS X) with "Mac Classic" theme. Note that add-cells-to-variable.rb is open in each editor.

To take advantage of syntax highlighting and coding support, we recommend installing and using one of the following programs for editing Ruby scripts:

- Windows:
    - Notepad++ (free and open-source)
    - Notepad2 (free and open-source)

- Mac OSX:
    - TextMate 2 (free and open-source)
    - gedit (free and open-source)
    - TextWrangler (free)

- Linux:
    - gedit (free and open-source)
    - kate (free and open-source)

General Principles

Each cell that you create with the Datavyu API has three inherent *codes*: onset, offset, and ordinal. Each cell also has at least one user-specified code.

onset, offset, and ordinal are all *Integers*, while the user-specified codes are *Strings*.

onset and offset are measured in **milliseconds from the beginning of the video**, starting from 0. For instance, an onset time of 00:02:20 translates to 140000ms.

Since user-specified codes are strings, you must convert any codes that you wish to perform calculations on to a numeric type. This is easily done with Ruby using the to_i method for integers, or the to_f method for floating point numbers.

Example

Create a variable, var1 whose value is "5". Since the number 5 has quotation marks around it, it is a string.

```
var1 = "5"
```

If you print var, you'd see that it is "5". Create a new variable, var2 from var1 using to_i to convert the "5" to a 5.

```
var2 = var1.to_i
```

Print var2 and see that it is a 5 without quotation marks:

```
print var2
```

```
p [5,6,7,8,9]
puts [5,6,7,8,9]
```

Basic Script Format

All Datavyu API scripts must include the following line at the top:

```
require 'Datavyu_API.rb'
```

This require statement loads all of the helper functions that enable your scripts to interact with the Datavyu spreadsheet.

In general, the rest of the script code goes between begin and end tags, making the general format as follows:

```ruby
require 'Datavyu_API.rb'
begin
  # Get the columns that we want to work with

  # Do something to those columns

  # Write any changes to those columns back to the spreadsheet
end
```

Tip: Anything that comes after a # character on a line in Ruby is a comment, which means it will not execute any specific task. Comments are useful for leaving notes that explain what the code is doing so that when you return to an old script you remember what you're looking at. The examples in this documentation use comments extensively.

Now that you are grounded in the Datavyu Ruby API mechanics, consider the *API Tutorials* or *Datavyu Ruby API Reference*.

2.2 Core Documentation

The Datavyu Ruby API's core documentation is organized into tutorials and reference pages. Tutorials provide step-by-step instructions for accomplishing common tasks with Ruby scripts. The Reference documentation provides a detailed reference for every *class* and *method* included in the API. This can be useful if you are uncertain about how to use a specific method or want an example of its use.

2.2.1 API Tutorials

Before You Start

Sample Data Most of the following tutorials use Example-Template.opf as their basic data file.
    The example template contains two *columns*: "id", whose codes describe basic information such as *subject number*, *test date*, *birth date*, and *condition*, and "trial", which has two codes: *trialnum* and *outcome*.
    The example has one *cell* in the "id" column and no cells in the "trial" column. Most of the Datavyu tutorials use the example template as their basis, so if you want to follow along directly, you can download it, and then open it with Datavyu (using File > Open and selecting the file from your hard drive).
    Most Datavyu scripts begin by loading a column from Datavyu using the getColumn() method. Before moving on to the larger tutorial list, you should familiarize yourself with loading Datavyu columns.

Load Datavyu Columns To load columns, Datavyu provides the getColumn() method. getColumn() takes one argument, the name of the column in Datavyu, and returns the Ruby representation of the column, which you can work with.
    The following script retrieves the "trial" column from the Datavyu spreadsheet and assigns it to an RColumn object called trial.

```ruby
require 'Datavyu_API.rb'
begin
  # Assign the Datavyu column "trial" data to a new Ruby object called trial.

  trial = getColumn("trial")
end
```

The left side of the method is the name of the Ruby object; the "trial" in the parentheses is the argument passed to getColumn() (the name of the Datavyu column).

Now that you're familiar with the sample data and know how to acquire data from Datavyu, you're ready to start scripting. The following tutorials will guide you through common tasks.

Tutorials

Add a New Column   Add a column to create a new coding pass or a new set of codes.

Add Codes to a Column   Add codes to a column to prompt coders which behaviors to score.

Add Cells to a Column   Add cells to a column while coding or annotating a video file.

Check for Coding Errors   Check for coding errors such as typos, impossible vlaues, unlikely values, etc.

Use Reliability Coding to Check Data Accuracy   Check Inter-rater Reliability to Improve Data Accuracy to determine whether more than one coder would score the data the same way.

Export Data Using Scripts   Use scripts to export data from Datavyu into a text file or a spreadsheet. for statistical analyses.

Convert an OpenShapa Script to Datavyu   Convert an OpenSHAPA script to the Datavyu format to update an old file.

Perform Operations on Multiple Files   Batch operations on multiple files to do the same operation on more than one file.

Convert a MacShapa file to Datavyu   Convert MacSHAPA files to work in Datavyu so that you can use the new, supported software.

Add a New Column   Much like *Add Cells to a Column* or *Add Codes to a Column*, the Datavyu API allows you to create a completely new column using the createColumn() method. createColumn() takes at least two parameters: first, the name of the new column, followed by a list of the names of the new codes contained in the column.

For example, suppose you wanted to add a column called "look" to your spreadsheet. And you wanted "look" to contain two codes: *direction* and *target*.

1. Set up the script, and create a new column with its two codes. You will need to create a Ruby object to hold the data until you are ready to write it back to the spreadsheet. In this example, the Ruby object is called look:

```
require 'Datavyu_API.rb'
begin
    # Create new column
    look = createColumn("look", "direction", "target")
```

2. Write the new column back to the spreadsheet and end the script:

```
require 'Datavyu_API.rb'
begin
    look = createColumn("look", "direction", "target")

    # Write the new column to Datavyu's spreadsheet.
    setColumn(look)
end
```

Add Codes to a Column Adding codes to a column is a straightforward scriptable task. The Datavyu API provides the add_code() method for adding codes.

add_code() takes the names of the Datavyu codes you are adding as its parameters. This example adds a code called *unit* to the "trial" Datavyu column in the sample data. The *unit* code might represent the unit of measure used during an experiment.

1. Start by setting up the script and assigning the Datavyu column "trial" to a variable using getColumn(). You can call your variable whatever you want to. We're calling it trial in this example:

```
require 'Datavyu_API.rb'
begin
    # Retrieve "trial" data from Datavyu spreadsheet and assign it
    # to a new Ruby variable
    trial = getColumn("trial")
```

2. Add the *unit* code to trial using add_code():

```
require 'Datavyu_API.rb'
begin
    trial = getColumn("trial")

    # Add the "unit" code to the trial variable
    trial.add_code("unit")
```

3. Write the changes back to the Datavyu spreadsheet using setColumn() and end the script:

```
require 'Datavyu_API.rb'
begin
    trial = getColumn("trial")
    trial.add_code("unit")

    # Write the changes back to the Datavyu spreadsheet
    setColumn(trial)
end
```

Add Cells to a Column Populating a Datavyu column with *cells* is another common task that you can automate with scripts.

Suppose that you want to code behaviors within 1-minute long blocks. Rather than have the coder manually insert each cell, you can write a script that will insert a "block" cell every minute, with the *onset* set to the correct trial start time.

1. Set up the script and create the Datavyu column you will be working with into a Ruby variable. We'll call it block here, but you can call it whatever you want to:

```
require 'Datavyu_API.rb'
begin
    # Creste the block column
    block = createColumn("block", "code")
```

2. Create the five cells using a loop.

Programming 101

Loops are types of code that tell Ruby to do something multiple times. Let's break down the for loop from the following example for those who aren't familiar with loops:

```
for i in 0..4
    <do stuff>
end
```

Essentially, this says, "let i = 0, <do stuff>, then, for i = 1, <do stuff>. Then again for i = 2, for i = 3, and for i = 4. Once i = 4, stop doing stuff." The 0..4 represents "from 0 to 4, inclusive".

If you wanted to do something ten times, your loop could read for i in 0..9 or for i in 1..10, or for i in 99..108.

In this case, you will use the value of i to set the *onset* time of each cell, so having it start at 0 and go to 4 makes sense.

Loop 5 times from 0 to 4, calculating the *onset* time to set for each new cell.

   a. Since Datavyu API uses times in milliseconds, convert i (the minute marker) to milliseconds:

```
require 'Datavyu_API.rb'
begin
   block = createColumn("block", "code")

   for i in 0..4
      # Calculate the onset time in milliseconds
      time = i * 1000 * 60
```

   b. Then, create the new cell using make_new_cell() and store the cell as a Ruby object. For simplicity, we'll call it cell:

```
require 'Datavyu_API.rb'
begin
    block = createColumn("block", "code")

   for i in 0..4
      time = i * 1000 * 60

      # Create a new cell, called ''cell''
      cell = block.make_new_cell()
```

   c. Set the onset of cell to the value of the time variable using change_code(), and end the for loop:

```
require 'Datavyu_API.rb'
begin
    block = createColumn("block", "code")

  for i in 0..4
    time = i * 1000 * 60
    cell = block.make_new_cell()

    # Set "onset" to the value of the "time" variable
    cell.change_code("onset", time)
  end
```

3. Now that the loop is complete, write the changes back to the Datavyu spreadsheet and end the script:

```
require 'Datavyu_API.rb'
begin
    block = createColumn("block", "code")

  for i in 0..4
    time = i * 1000 * 60
    cell = block.make_new_cell()
    cell.change_code("onset", time)
  end

  # Write change back to the Datavyu spreadsheet
  setColumn(block)
end
```

Check for Coding Errors Datavyu scripts have the ability to check for coding errors to make up for potential human error. Inputting invalid codes is a common coding mistake. For example, a coder might accidentally input an "h", when only "j" or "k" are acceptable values. Using scripts, coders can double-check their work for errors and fix them early on.

Datavyu API provides the checkValidCodes() method to check for coder errors. checkValidCodes() requires at least three arguments: the name of the column it will verify, the location that it will output the results to, and at least one *key-value pair*. Key-value pairs consist of a "key", the name of a Datavyu code, and a "value", which is an array of valid values for that code. checkValidCodes() then checks each code (key) against its list of valid values (the values).

The following examples check the look column against its codes. You can download the sample data used in this tutorial from here.

Check Code Validity and Output to the Console

Basic Format
Example

```
require 'Datavyu_API.rb'
begin
  # Check for errors. Notice the square brackets. These denote
  # lists. The basic format is: "columnName", "dumpFile",
  # "codename", ["validcode1", "validcode2", etc],
  # "code2", [ "validcode1", "validcode2", etc], ...
```

```
    checkValidCodes("look", "", "direction", ["l", "r"], "target", ["a", "b"])
end
```

Breaking down the function call makes it easier to follow what is happening. Recall that checkValidCodes() takes (at least) three arguments: the name of the Datavyu variable it will be checking, the location that it should direct the output to, and at least one key-value pair.

In the example, we have:

```
checkValidCodes("look", "", "direction", ["l", "r"], "target", ["a", "b"])
```

- "look" is the name of the variable to check.

- "" is the location that we want to export the output to. Using "" indicates that we do not want to write the results to a file, and that it should instead display in the Datavyu Scripting Console.

- "direction", ["l", "r"] is the first key-value pair, which specifies that the "direction" Datavyu code should only have the values "l" or "r".

- "target" , ["a", "b"] is the second key-value pair, which specifies that the "target" Datavyu code should only have the values "a" or "b".

Advanced Format   You can perform the same verification by first assigning the valid codes to objects. This makes it is easier for human readers to parse the script, and makes it easy to modify or update in the future.

1. Set up the script:

```
require 'Datavyu_API.rb'
begin
```

2. Assign each list of valid codes to a variable:

```
require 'Datavyu_API.rb'
begin
    # Store each of the valid code arrays into a object first
    # so that it is easier to read
    directionCodes = ["l", "r"]
    targetCodes = ["a", "b"]
```

3. Check for coding errors using checkValidCodes(), replacing the lists with your newly-created objects, and end the script:

```
require 'Datavyu_API.rb'
begin
    directionCodes = ["l", "r"]
    targetCodes = ["h", "t"]

    # Check for typos, replacing the code arrays with your new variables:
    checkValidCodes("look", "", "direction", directionCodes, "target", targetCodes)
end
```

38

Check Code Validity and Output to a File checkValidCodes() can write the results of its verification to a "dump file". Datavyu will create an output file if you specify a path in the dumpFile parameter.

1. Create an object that holds the path that it should output to using Ruby's File.expand_path method, which converts a relative path, like ~/Desktop/file.txt to an absolute path name, which contains the root directory, and all sub-directories, like /Users/alice/Desktop/file.txt

   The following commands create a variable called output that holds the absolute path to a file on the Desktop called output.txt:

```
require 'Datavyu_API.rb'
begin

    output = File.expand_path("~/Desktop/output.txt")
```

2. Call checkValidCodes() on the "step" column, passing output as the argument for the dumpFile parameter:

```
require 'Datavyu_API.rb'
begin
    directionCodes = ["l", "r"]
    targetCodes = ["a", "b"]

    output = File.expand_path("~/Desktop/output.txt")

    # Check for errors, specifying the output variable as the dumpFile parameter
    checkValidCodes("look", output, "direction", directionCodes, "target", targetCodes)
end
```

When the script ends, the output.txt file will be created on the Desktop, containing the results of the code checking. For the sample data, it should find one error, and the output should resemble:

```
Code ERROR: Var: look    Ordinal: 2      Arg: direction      Val: rj
```

Video Example of Checking for Errors This video displays one way to check for errors (typos, impossible values, etc.) within a spreadsheet.

Check Inter-rater Reliability to Improve Data Accuracy Reliability coding verifies that the desired codes are observable and that the coders are accurately interpreting events. With reliability coding, two coders separately code the same data source. You can create a *reliability column*, which the second coder can use to reccord his or her observations. After both coders have coded the same column, they can compare their codes and determine their inter-rater reliability.

Note: While it may be tempting to have the reliability coder code the data source in a new spreadsheet, you should endeavor to keep all codes for a given data source in a single spreadsheet. This facilitates analysis and ensures that like data are kept together. To prevent the reliability coder from seeing the original coding pass, you can *hide the original coded column*.

Make a Reliability Column In general, when creating reliability columns you create blank cells that correspond to the cells created during the first coding pass. This ensures that the two coders will have observed the same events in the data stream and allows easy comparison of the two coding passes.

The Datavyu Ruby API provides the makeReliability() method for creating reliability variables. makeReliability() has four parameters:

Parameter	Type	Description
relname	String **or** Ruby column from getColumn()	the name of the new reliability column you will create. The convention is to name it rel_columnName, but you can name it whatever you want to.
column_to_copy	String	The name of the column that we want to create a reliability column *from* (i.e. the existing coded column).
multiple_to_keep	Integer (**optional**)	Number of cells to skip: a value of 2 includes *every other cell* in the new variable; 1 includes every cell, and 0 creates a blank column with no cells
*codes_to_keep	comma-separated strings (**optional**)	Codes you want to copy from the original to new column. These are codes that the reliability coder will not have to code.

Tip: Copying the *onset* of the original column to the new reliability column in the args_to_keep parameter makes it easier for the second coder to navigate to the correct locations in the data source, and to code the same events as the original coder.

When making a reliability column, you should also think about how you are going to compare the columns. checkReliability(), which you use to check reliability. It requires that each pair of cells has a unique identifier that link them together. For example, a trial number coded into each cell would match corresponding cells, even if only a subset of cells were included in the reliability variable.

The following example uses the sample data to create a new reliability column called "trial_rel" from its "trial" column, skipping every other cell, and copying over the *onset* and *trialnum* codes so that the reliability coder doesn't have to recode *onset* and trial numbers.

```ruby
require 'Datavyu_API.rb'
begin
  makeReliability("trial_rel", "trial", 2, "onset", "trialnum")
end
```

Note: You do **not** have to write the variable back to the spreadsheet. makeReliability() automatically writes its results to the spreadsheet.

Check Reliability Once the second coder has recorded their observations in the reliability column, you can use checkReliability() to compare the primary and reliability columns cells. checkReliability() returns the number of errors, and the percent agreement.

checkReliability() has four required parameters, and one optional one:

Parameter	Type	Description
main_col	String **or** Ruby variable from getColumn()	The primary column that rel_col will be compared against
rel_col	String **or** Ruby variable from getColumn()	The reliability column to compare to main_col
match_arg	String	The argument used to match the reliability cells to the primary cells. Must be a unique identifier between the cells.
time_tolerance	Integer	Amount of slack permitted, in milliseconds, between the two onset and offsets before it will be considered an error. Set to 0 to tolerate no difference.
dump_file	String path or Ruby File object (**optional**)	The full string path to dump the reliability output to. This can be used for multi-file dumps or just to keep a log. You can also give it a Ruby File object if a file is already started.

Note: match_arg is particularly important: for checkReliability() to know which cells to compare, it needs to have some parameter that is unique to each pair of corresponding primary and reliability cells. In many cases, the onset time of the cell can be used to match primary and reliability cases.

1. Create an object that holds the path that it should output to using Ruby's File.expand_path method, which converts a relative path, like ~/Desktop/file.txt to an absolute path name, which contains the root directory, and all sub-directories, like /Users/alice/Desktop/file.txt

   The following commands create a variable called dump_file that holds the absolute path to a file on the Desktop called relcheck.txt:

```
require 'Datavyu_API.rb'
begin

    dump_file = File.expand_path("~/Desktop/relCheck.txt")
```

\#. Compare "trial" and "**rel_** " using checkReliability(),

   with a 5ms time tolerance, and output the results to the dump_file:

```
require 'Datavyu_API.rb'
begin
    dump_file = File.expand_path("~/Desktop/relcheck.txt")

    # Compare the "trial" and "trial_rel" columns, using trialnumber as
    #  their matching code and dump the results to a file on the desktop.
    checkReliability("trial", "trial_rel", "trialnum", 5, dump_file)
end
```

Video Example of Checking for Reliability   This video displays one way to check for inter-rater reliability for a single column in a spreadsheet.

Use Scripts to Export Data from Datavyu

Export Methods   Datavyu supports numerous data export methods. The *Export Data from Datavyu* section of the software guide tutorials demonstrates how to export data to a basic .csv file using the *Export File* function. The Ruby scripting API offers users more flexibility in specifying different output file formats for exporting data. This section covers two scripting approaches:

- A straight *frame-by-frame* dump detailing all observations associated with each frame.

- A *nested* export, which loops through each column and nests cells appropriately.

If you wish to export data from multiple files, refer to the *Batch Operations on Multiple Files* tutorial for guidance on operations that involve multiple files.

Method : Frame-by-Frame Export   A frame-by-frame export prints a row for every video frame in the spreadsheet and looks across every column and code and writes the values for that frame. This method is particularly useful for free-form coding projects that contain multiple columns that do not nest in time. Using a frame-by-frame export makes it easy to import your data into other software packages (e.g., Excel, SPSS, R).

Datavyu includes a script that performs frame-by-frame export on any file automatically. By default, the script "Export Data by Frame" will appear in the "favorites" folder in Datavyu. To test the script, open the sample spreadsheet in Datavyu. Once the spreadsheet has loaded, select the Script menu and then select "Favorites/Export Data by Frame.rb". That's it! The script will output a .csv file to your desktop called "framebyframe_export.csv" that contains all of the data from the spreadsheet that can be opened in the statistical package of your choice.

If you would like to export multiple files frame-by-frame, there is a script included for that as well ("Export Data by Frame – Multiple.rb"). Simply create a folder on your desktop called "datavyu_files" and

time	MomSpeech.ordinal	MomSpeech.onset	MomSpeech.offset	MomSpeech.transcript	MomObject.ordinal	MomObject.onset	MomObject.offset	MomObject.object1
84414	14	83605	85081	[sipping sounds]	6	80780	85734	cup
84447	14	83605	85081	[sipping sounds]	6	80780	85734	cup
84480	14	83605	85081	[sipping sounds]	6	80780	85734	cup
84513	14	83605	85081	[sipping sounds]	6	80780	85734	cup
84546	14	83605	85081	[sipping sounds]	6	80780	85734	cup
84579	14	83605	85081	[sipping sounds]	6	80780	85734	cup
84612	14	83605	85081	[sipping sounds]	6	80780	85734	cup
84645	14	83605	85081	[sipping sounds]	6	80780	85734	cup
84678	14	83605	85081	[sipping sounds]	6	80780	85734	cup
84711	14	83605	85081	[sipping sounds]	6	80780	85734	cup
84744	14	83605	85081	[sipping sounds]	6	80780	85734	cup
84777	14	83605	85081	[sipping sounds]	6	80780	85734	cup
84810	14	83605	85081	[sipping sounds]	6	80780	85734	cup
84843	14	83605	85081	[sipping sounds]	6	80780	85734	cup
84876	14	83605	85081	[sipping sounds]	6	80780	85734	cup
84909	14	83605	85081	[sipping sounds]	6	80780	85734	cup
84942	14	83605	85081	[sipping sounds]	6	80780	85734	cup
84975	14	83605	85081	[sipping sounds]	6	80780	85734	cup
85008	14	83605	85081	[sipping sounds]	6	80780	85734	cup
85041	14	83605	85081	[sipping sounds]	6	80780	85734	cup
85074	14	83605	85081	[sipping sounds]	6	80780	85734	cup
85107					6	80780	85734	cup
85140					6	80780	85734	cup
85173	15	85155	87645	Where's your cup?	6	80780	85734	cup
85206	15	85155	87645	Where's your cup?	6	80780	85734	cup
85239	15	85155	87645	Where's your cup?	6	80780	85734	cup
85272	15	85155	87645	Where's your cup?	6	80780	85734	cup
85305	15	85155	87645	Where's your cup?	6	80780	85734	cup
85338	15	85155	87645	Where's your cup?	6	80780	85734	cup

place the files you want to export in that folder. Please note that the files should contain the same columns and codes to export correctly.

These scripts will work on a variety of files and may fit many users' needs. However, if you want to tailor the scripts for your own purposes (e.g., changing the output file, input folder, or delimiter), you can find the script files in your Datavyu installation folder under the "Favorites" folder.

Method : Nested Export   A *nested export* exports data based on the nesting of *cells*. This is most useful for spreadsheets whose cells group together. For instance, the following spreadsheet example has three columns: "id", "trial", and "foot". "id" is a participant ID, which might include codes that describe the participant's individual id code, gender, age, etc., "trial" is a column that marks each trial that occurred and "foot" is a column representing observations recorded during each trial.

The cells, then, group together with an "id" cell covering the length of all trials. There are two trials in the example that occur within the time limits of the "id" cell and there are several "foot" cells that occur within the time limits of each trial.

To export data from this style of spreadsheet, we will use a series of *loops*, exporting a row of tab-separated values for each cell in the "foot" column. Each row will include the "id" and "trial" data that the "foot" cell nests beside. This spreadsheet has only one "id" cell, since all of the data in that spreadsheet is for a single participant.

1. Set up the script, and then define where you are going to output the file to. You need to define the location of the file (in this case, the Desktop, defined by out_file), and create a Ruby object to hold the new file's data as it outputs it, which we'll call out:

```ruby
require 'Datavyu_API.rb'

begin


  # Defines the location of the file that we're going to be outputting
  # the spreadsheet data to - the file name is DataOutput.txt
  # and is located on the Desktop.
  out_file = File.expand_path("~/Desktop/DataOutput.txt")
```

id (MATRIX)	trial (MATRIX)	foot (MATRIX)	+	
1　　00:00:00:000 00:06:50:312 (9999, m)	1　　00:00:00:000 00:04:38:504 (127, 9/23/2013)	1　　　00:00:02:142 00:00:02:346 (l)		
		2　　　00:00:21:216 00:00:22:338 (l)		
		3　　　00:00:46:682 00:00:46:920 (r)		
		4　　　00:01:09:564 00:01:09:904 (r)		
		5　　　00:01:13:270 00:01:13:746 (l)		
		6　　　00:02:32:660 00:02:32:762 (r)		
		7　　　00:02:35:108 00:02:35:244 (l)		
		8　　　00:02:58:296 00:02:59:010 (r)		
		9　　　00:03:01:118 00:03:01:696 (l)		
		10　　00:03:01:764 00:03:02:444 (l)		
	2　　00:05:25:686 00:06:50:312 (128, 9/23/2013)	11　　00:05:25:686 00:05:25:992 (l)		
		12　　00:05:26:706 00:05:26:910 (l)		
		13　　00:05:43:298 00:05:43:400 (r)		
		14　　00:05:44:148 00:05:44:284 (l)		
		15　　00:06:27:328 00:06:27:430 (r)		
		16　　00:06:27:838 00:06:27:906 (r)		
		17　　00:06:28:008 00:06:28:144 (l)		
		18　　00:06:30:660 00:06:30:762 (r)		
		19　　00:06:34:468 00:06:34:536 (l)		
		20　　00:06:34:672 00:06:35:590 (r)		

```
# Creates the file, and assigns write permissions so that the system
# can write to it ('w')
out = File.new(out_file,'w')
```

2. Retrieve the columns you want to output from the spreadsheet, and assign them to RColumn objects:

```
require 'Datavyu_API.rb'

begin

    out_file = File.expand_path("~/Desktop/DataOutput.txt")
    out = File.new(out_file,'w')

    # Retrieve the "id", "trial" and "foot" columns from the spreadsheet
    # and assign them to RColumn objects

    id = getColumn("id")
    trial = getColumn("trial")
    foot = getColumn("foot")
```

3. Set up a series of for loops that we will use to iterate over each cell in the columns we're interested in:

```
require 'Datavyu_API.rb'

begin

    out_file = File.expand_path("~/Desktop/DataOutput.txt")
    out = File.new(out_file,'w')

    id = getColumn("id")
    trial = getColumn("trial")
    foot = getColumn("foot")

    # Set up a series of nested for loops, following the nesting
    # of the cells: "id", then "trial", then "foot".
    #
    # This will iterate through every "cell"
    # in id, every cell in "trial", and every cell in "foot".
    #
    # idcell, tcell, and fcell are temporary Ruby variables
    # that hold the data for a cell as the cell is iterated over.


    for idcell in id.cells
        for tcell in trial.cells
            for fcell in foot.cells
```

4. Write an if clause that checks if cells are nested. In plain English, this if statement checks if the *onset* of the foot cell (fcell) occurs after the onset of the trial cell (tcell) and that the *offset* of the fcell occurs *before* the offset of the tcell. Or, in even plainer English, that the fcell occurs during the length of the tcell.

If the clauses are met, write the cells' codes to the out file, separated by tabs:

```ruby
require 'Datavyu_API.rb'

begin

  out_file = File.expand_path("~/Desktop/DataOutput.txt")
  out = File.new(out_file,'w')

  id = getColumn("id")
  trial = getColumn("trial")
  foot = getColumn("foot")

  for idcell in id.cells
    for tcell in trial.cells
      for fcell in foot.cells

        # Set up if statement that checks for cell encapsulation

        if lcell.onset >= tcell.onset && lcell.offset <= tcell.offset
```

5. If the if clause is met, write the cells' codes to the out file, separated by tabs.

   As a refresher, cells have a series of codes: onset, offset, and ordinal by default, as well as any user-specified codes. To access the codes in one of our temporary Ruby variables (icell, tcell, and fcell), we use the format cellName.codeName. To access the "idnum" code in the "id" column, then, we'd request icell.idnum.

   Since the script is outputting *strings*, we also need to convert the onset, offset, and ordinals from the *integer* format to the string format, using to_s.

```ruby
require 'Datavyu_API.rb'

begin

  out_file = File.expand_path("~/Desktop/DataOutput.txt")
  out = File.new(out_file,'w')

  id = getColumn("id")
  trial = getColumn("trial")
  foot = getColumn("foot")

  for idcell in id.cells
    for tcell in trial.cells
      for fcell in foot.cells
        if fcell.onset >= tcell.onset && fcell.offset <= tcell.offset
          # Write the cells' codes to the output file, separated by tabs - the "\t"
          # You must include a newline indicated, "\n" so that the next cells' codes
          # will be output on a new line, giving them their own row.
          out.write(idcell.idnum + "\t" + tcell.onset.to_s + "\t" +
            tcell.offset.to_s + "\t" + tcell.trial + "\t" +
            fcell.ordinal.to_s + "\t" + fcell.onset.to_s + "\t" +
            fcell.offset.to_s + "\t" + fcell.side + "\n")
          # End the if clause, and the for loops, as well as the script
        end
      end
```

```
      end
    end

end
```

Video Example of Running an Export Script   This video displays a user running a script to export data in a specific way. This export script exports all of the columns of one spreadsheet into an Excel file. Datavyu's built-in export displays one Excel cell as one cell on the spreadsheet. This specific script repeats all the participant information such as the id number, test date, birthdate, and sex, for all of the data making it more useful when you import into a statistical analysis program.

Convert an OpenSHAPA Script to the Datavyu Format   If you have previously written scripts for use with OpenSHAPA, you can easily reuse them with Datavyu.

Datavyu's Ruby API was designed to be compatible with the earlier scripting formats. To update scripts from OpenSHAPA, you simply need to delete the OpenSHAPA API code at the beginning of the file and replace it with the following line of code:

```
require 'Datavyu_API.rb'
```

And with that, you should be able to run your existing OpenSHAPA scripts through Datavyu.

Batch Operations on Multiple Files   Writing scripts that act on multiple files is largely the same as writing scripts that act on a single file: the interactions with columns, codes, and cells are the same. The scripting API includes commands that can load and save Datavyu spreadsheets. After a spreadsheet is loaded, the script can execute commands on the spreadsheet (e.g., export data, change or add columns), then open the next spreadsheet, execute commands, and so on.

This tutorial loads the spreadsheet files in a folder called datafiles that is located on the Desktop, accesses the "id", "trial" and "foot" columns, and then sets up a loop to export the data, following the process in *Use Scripts to Export Data from Datavyu*.

1. Create the output data file that you will be exporting the spreadsheet data to:

```
out_file = File.expand_path("~/Desktop/DataOutput.txt")
out = File.new(out_file,'w')
```

2. Identify the folder that contains the spreadsheets whose data you want to export, and create a list object called filenames that lists the files in that folder:

```
out_file = File.expand_path("~/Desktop/DataOutput.txt")
out = File.new(out_file,'w')

# Locate the files in the datafiles folder on the Desktop
# and assign the list of file names to the filenames Ruby
# object.
filedir = File.expand_path("~/Desktop/datafiles/")
filenames = Dir.new(filedir).entries
```

3. Iterate through each file in filenames to see if it contains data and is a .opf Datavyu spreadsheet file. If so, load the spreadsheet data into Datavyu, and print "SUCCESSFULLY LOADED" when complete:

```
out_file = File.expand_path("~/Desktop/DataOutput.txt")
out = File.new(out_file,'w')
```

```
filedir = File.expand_path("~/Desktop/datafiles/")
filenames = Dir.new(filedir).entries

# Iterate through each filename in the "filenames" list
for file in filenames
   if file.include?(".opf") and file[0].chr != '.'

   puts "LOADING DATABASE: " + filedir+file
   $db,proj = load_db(filedir+file)
   puts "SUCCESSFULLY LOADED"
```

4. Get the columns you are going to export using getColumn():

```
out_file = File.expand_path("~/Desktop/DataOutput.txt")
out = File.new(out_file,'w')
filedir = File.expand_path("~/Desktop/datafiles/")
filenames = Dir.new(filedir).entries

for file in filenames
   if file.include?(".opf") and file[0].chr != '.'

   puts "LOADING DATABASE: " + filedir+file
   $db,proj = load_db(filedir+file)
   puts "SUCCESSFULLY LOADED"

   # Retrieve "id", "trial", and "foot" columns

   id = getColumn("id")
   trial = getColumn("trial")
   foot = getColumn("foot")
```

5. Set up a series of for loops to iterate over each cell in the relevant columns, and then use an if to export nested cells, following the steps in *Use Scripts to Export Data from Datavyu*:

```
out_file = File.expand_path("~/Desktop/DataOutput.txt")
out = File.new(out_file,'w')
filedir = File.expand_path("~/Desktop/datafiles/")
filenames = Dir.new(filedir).entries

for file in filenames
   if file.include?(".opf") and file[0].chr != '.'

   puts "LOADING DATABASE: " + filedir+file
   $db,proj = load_db(filedir+file)
   puts "SUCCESSFULLY LOADED"

   id = getColumn("id")
   trial = getColumn("trial")
   foot = getColumn("foot")

   # Export Data
```

```
    for idcell in id.cells
        for tcell in trial.cells
            for fcell in foot.cells
                if fcell.onset >= tcell.onset && fcell.offset <= tcell.offset
                    # Write the cells' codes to the output file, separated by tabs - the "\t"
                    # You must include a newline indicated, "\n" so that the next cells' codes
                    # will be output on a new line, giving them their own row.
                    out.write(idcell.idnum + "\t" + tcell.onset.to_s + "\t" +
                        tcell.offset.to_s + "\t" + tcell.trial + "\t" +
                        fcell.ordinal.to_s + "\t" + fcell.onset.to_s + "\t" +
                        fcell.offset.to_s + "\t" + fcell.side + "\n")
                    # End the if clause, and the for loops, as well as the script
                end
            end
        end
    end
```

6. Close the filename for loop and if statement and print "FINISHED" when the script has finished exporting all the spreadsheet files' data to the output file:

```
require 'Datavyu_API.rb'
 begin

    out_file = File.expand_path("~/Desktop/DataOutput.txt")
    out = File.new(out_file,'w')
    filedir = File.expand_path("~/Desktop/datafiles/")
    filenames = Dir.new(filedir).entries

    for file in filenames
        if file.include?(".opf") and file[0].chr != '.'

        puts "LOADING DATABASE: " + filedir+file
        $db,proj = load_db(filedir+file)
        puts "SUCCESSFULLY LOADED"

        id = getColumn("id")
        trial = getColumn("trial")
        foot = getColumn("foot")

        # Export Data

        for idcell in id.cells
            for tcell in trial.cells
                for fcell in foot.cells
                    if fcell.onset >= tcell.onset && fcell.offset <= tcell.offset

                        # Write the cells' codes to the output file, separated by tabs - the "\t"
                        # You must include a newline indicated, "\n" so that the next cells' codes
                        # will be output on a new line, giving them their own row.
                        out.write(idcell.idnum + "\t" + tcell.onset.to_s + "\t" +
                            tcell.offset.to_s + "\t" + tcell.trial + "\t" +
                            fcell.ordinal.to_s + "\t" + fcell.onset.to_s + "\t" +
```

```ruby
                    fcell.offset.to_s + "\t" + fcell.side + "\n")
                  # End the if clause, and the for loops, as well as the script
                  end
              end
            end
          end

        end
    end

    puts "FINISHED!"

    end

# End the script
end
```

Convert MacSHAPA Files to Work in Datavyu   MacSHAPA users can convert their files to the new Datavyu format using a script. The following script converts a folder of MacSHAPA files to Datavyu files, but be sure to edit your folders' names and locations to reflect the location of your MacSHAPA files and Datavyu files.

```ruby
require 'Datavyu_API.rb'
begin

  # Edit this to match the directory containing your files.
  macshapa_folder = File.expand_path("~/Desktop/MacSHAPA/")
  macshapa_files = Dir.new(macshapa_folder)

  # Edit this to match where you want to save the datavyu files.
  datavyu_folder = File.expand_path("~/Desktop/Datavyu/")
  if (!File::directory?(datavyu_folder))
      Dir.mkdir(datavyu_folder) # Make this dir if it doesn't exist
  end

  for f in macshapa_files.each()
   # Filter out files we don't want
      if (f[0].chr != '.') # Filter out the hidden files like . and .. and .DSSTORE
          puts "Converting " + f
          # Load the file and don't draw it to the screen
          $db, proj = load_macshapa_db(macshapa_folder + '/' + f, false)
          puts "Saving file " + f + " as Datavyu file."
          save_db(datavyu_folder + '/' + f + ".opf")
      end
  end
end
```

2.2.2   Datavyu Ruby API Reference

Version 1.3.4

Classes and Class Methods

CTable Class

class CTable

Represent a contingency table / confusion matrix for a single code.

classmethod add(*pri_value*, *rel_value*)

Increment the table value at the given combination by one. See computeKappa() for automatically computing kappa scores.

Parameter	Type	Description
pri_value	String	Value for primary coder.
rel_value	String	Value for reliability coder.

Returns

None.

classmethod ef(*idx*)

Return the expected frequency of agreement by chance for the given index.

Parameter	Type	Description
idx	Integer	Index of code (starting at zero).

classmethod efs()

Return the sum of the expected frequency of agreement by chance for all indices in table.

classmethod kappa()

Compute kappa using table values.

classmethod total()

Return the sum of all elements in table.

classmethod to_s()

Return formateed string to display the table.

RCell Class

class RCell

The Ruby container for Datavyu cells.

classmethod change_code(*code*, *val*)

Changes the value of a code in a cell.

Parameter	Type	Description
arg	String **or** Ruby column from getColumn()	Name of the code that you are updating.
val	String, Integer, etc.	Value to change the code to.

Returns

None.

Example

The following example sets the "trial" column's cell at position 0's *onset* to 1000ms, and then writes the change back to the spreadsheet using setColumn().

```
require 'Datavyu_API.rb'
begin
  trial = getColumn("trial")
  trial.cells[0].change_code("onset", 1000)
  setColumn("trial", trial)
end
```

classmethod is_within(*outer_cell*)

Determines if a cell is temporally encased by the outer_cell.

[t]|l|L|L|

Parameter	Type	Description
outer_cell		The cell that is going to be checked to see if it temporally encases the study cell.

Returns

Boolean

Example

Compare the first cell of the "trial" and "id" columns to see if the first cell of "trial" is temporally enclosed by the first cell of "id". If it is, print out "Yes, it is temporally enclosed", otherwise, print "No, it is not temporally enclosed."

```
require 'Datavyu_API.rb'
begin
  trial = getColumn("trial")
  id = getColumn("id")
  if trial.cells[0].is_within(id.cells[0])
    puts "Yes, it is temporally enclosed."
  else
    puts "No, it is not temporally enclosed."
  end
end
```

classmethod contains(*inner_cell*)

Determines if a cell temporally encases the innter_cell.

[t]|l|L|L|

Parameter	Type	Description
inner_cell		The cell that is going to be checked to see if it is temporally encased by the study cell.

Returns

Boolean

Example

Compare the first cell of the "trial" and "id" columns to see if the first cell of "trial" is temporally enclosed by the first cell of "id". If it is, print out "Yes, it is temporally encloses the cell", otherwise, print "No, it is does not temporally enclose the cell."

```
require 'Datavyu_API.rb'
begin
  trial = getColumn("trial")
  id = getColumn("id")
  if id.cells[0].is_within(trial.cells[0])
    puts "Yes, it is temporally encloses the cell."
  else
    puts "No, it is does not temporally enclose the cell."
  end
end
```

classmethod print_all(*p)

Dumps all of the codes in a *cell* to a *string*.

Parameter	Type	Description
p **optional**	String	The separator between codes. Defaults to tab (t)

Returns

String of the codes, starting with ordinal, onset, and offset, followed by the codes.

Example

The following example prints all of the "trial" column's first cell's codes using print.

```
require 'Datavyu_API.rb'
begin
  trial = getColumn("trial")
  print trial.cells[0].print_all()
end
```

RColumn Class

class RColumn

The Ruby container for Datavyu columns.

classmethod make_new_cell()

Creates a new blank cell at the end of the column's cell array.

Argument	Type	Description
None		

Returns

Reference to the cell that was just created. Modify the cell using this reference.

Example

The following example creates a new cell at the end of the "trial" column's cell array and assigns its reference to the variable newcell. It then changes newcell's *onset* to 1000ms using change_code() and writes the change back to the spreadsheet using setColumn().

```
trial = getColumn("trial")
newcell = trial.make_new_cell()
newcell.change_code("onset", 1000)
setColumn("trial", trial)
```

classmethod change_code_name(*old_name*, *new_name*)

Renames a code.

[t]|l|L|L|

Argument	Type	Description
old_name	String	Current name of the code
new_name	String	New name for the code, which will replace old_name

Returns

Nothing.

Example

The following example renames the "trial" column's *bad_code_name* code to *awesome_code_name* and then writes the changes back to the Datavyu spreadsheet:

```
require 'Datavyu_API.rb'
begin
  trial = getColumn("trial")
  trial.change_code_name("bad_code_name", "awesome_code_name")
  setColumn("trial", trial)
end
```

classmethod add_code(*name*)

Adds a code to a column.

[t]|l|L|L|

Argument	Type	Description
name	String	The name of the code you are adding to the column

Returns

Nothing.

Example

The following example adds the *unit* code to the "trial" column and then writes the changes back to the spreadsheet using setColumn().

```
require 'Datavyu_API.rb'
begin
  trial = getColumn("trial")
  trial.add_code("unit")
  setColumn(trial)
end
```

classmethod remove_code(*name*)

Deletes a code from a column.

[t]|l|L|L|

Argument	Type	Description
name	String	The name of the code you wish to delete from the column

Returns

Nothing.

Freestanding Methods

add_codes_to_column()

add_codes_to_column(*column, *codes*)

Add new codes to a column.

[t]|l|L|L|

Parameter	Type	Description
column	String or RColumn object	Name of the variable that you are adding arguments to.
*codes	List	List of arguments to add to column. Parameters with an * indicate lists.

Returns

Ruby representation of the column.

Example

The following example adds "code1", "code2", and "code3" to the "test" column, and writes it back to the spreadsheet using setColumn().

```
require 'Datavyu_API.rb'
begin
  test = add_codes_to_column("test", "code1", "code2", "code3")
  setColumn("test", test)
end
```

checkReliability()

checkReliability(*main_col, rel_col, match_arg, time_tolerance, dump_file*)

Compares two Datavyu columns to check for reliability errors and accuracy.

[t]|l|L|L|

Parameter	Type	Description
main_col	String **or** Ruby variable from getColumn()	The primary column that rel_col will be compared against
rel_col	String **or** Ruby variable from getColumn()	The reliability column to compare to main_col
match_arg	String	The argument used to match the reliability cells to the primary cells. Must be a unique identifier between the cells.
time_tolerance	Integer	Amount of slack permitted, in milliseconds, between the two onset and offsets before it will be considered an error. Set to 0 to tolerate no difference.
dump_file	String path or Ruby File object (**optional**)	The full string path to dump the reliability output to. This can be used for multi-file dumps or just to keep a log. You can also give it a Ruby File object if a file is already started.

Returns

Console and file output.

Example

The following example checks the reliability column "rel_trial" against the primary column "trial", linking the two on their "trialnum" code, with a 100ms *onset* and *offset* difference tolerated.

checkReliability("trial", "rel_trial", "trialnum", 100)

Example

The following example performs the same operation as the previous example, but also writes the output to ~/Desktop/Relcheck.txt, a text file.

checkReliability("trial", "rel_trial", "trialnum", 100, "~/Desktop/Relcheck.txt")

See also:
Check Inter-rater Reliability to Improve Data Accuracy

checkValidCodes()
checkValidCodes(*column*, *dump_file*, **arg_code_pairs*)

Checks that all coded values in Datavyu conform to a the list of valid codes for that column.

Argument	Type	Description
column	String	The Datavyu column that to check
dump_file	String, or Ruby File object	Full path of the file to dump output to. Use "" to write to the console. You can also specify a Ruby File object.
*arg_code_pairs	Key-value pairs	List of code names and valid values, in the format "code_name", ["valid1", "valid2"], "code_name2", ["valid3", "valid4"], etc.

Returns

Console and/or file input.

Example

The following example checks the validity of the codes for the "trial" Datavyu column:

check_valid_codes("trial", "", "hand", ["l","r","b","n"], "turn", ["l","r"],
    "unit", ["1","2","3"])

See also:
Check for Coding Errors

combine_columns()

combine_columns(*name*, **columnNames*)

Combines two column together, creating a new column. create_mutually_exclusive() combines the two source columns cells together so that the new column includes all of the arguments from both source columns. It also includes a combination of the two columns' cells as well as a new cell for each overlap.

Parameter	Type	Description
name	String	The name of the new column you wish to create
columnNames	List of strings	The names of the source column that will be combined

Returns

The new Ruby representation of the column. You must write the changes back to the spreadsheet if you want to save them.

Example

The following example creates a column called "test" from two existing columns, "col1" and "col2", and then writes the changes back to the spreadsheet using setColumn().

```ruby
require 'Datavyu_API.rb'
begin
  test = combine_columns("test", "col1", "col2")
  setColumn("test", test)
end
```

computeKappa()

computeKappa(*pri_col*, *rel_col*, **codes*)

Computes Cohen's kappa for a primary and reliability column. Cells between the two columns are matched by their onset time. Computes a contingency table and kappa score for each specified code.

Parameter	Type	Description
pri_col	String or RColumn object	Name (or column object) of primary coder's column.
rel_col	String or RColumn object	Name (or column object) of reliability coder's column.
*codes	List	Names of codes to compute scores for.

Returns

Hashes (associative arrays) for kappa values and CTable, in that order. Keys are names of the codes. Values are Numeric, and :class`Ctable`, respectively.

createColumn()

createColumn(*name*, **codes*)

Creates a new blank column with the specified name and codes.

Parameter	Type	Description
name	String	The name of the new column
codes	comma separated Strings	Codes that the new column will contain.

Note: createColumn() creates the onset, offset, and ordinal codes by default. You do not need to specify them in the codes.

Returns

Ruby object representation of the new column in Datavyu.

Example

The following example creates a new Datavyu column called "trial" with the codes "trialnum" and "unit", and assigns them to an RColumn object called trial. It then adds a new cell to trial using make_new_cell() and writes the changes back to the Datavyu spreadsheet using setColumn().

```
require 'Datavyu_API.rb'
begin
  trial = createColumn("trial", "trialnum", "unit")
  trial.make_new_cell()
  setColumn(trial)
end
```

---

create_mutually_exclusive()

create_mutually_exclusive(*name, col1name, col2name*)

Combines two column together, creating a new column.:func:*create_mutually_exclusive* combines the two source columns cells together so that the new column includes all of the arguments from both source columns. It also includes a combination of the two columns' cells as well as a new cell for each overlap.

[t]|l|L|L|

Parameter	Type	Description
name	String	The name of the new column you wish to create.
col1name	String	The name of the first source column to combine.
col2name	String	The name of the second source column to combine.

Returns

The new Ruby representation of the column. You must write the changes back to the spreadsheet if you want to save them.

Example

The following example creates a column called "test" from two existing columns, "col1" and "col2", and then writes the changes back to the spreadsheet using setColumn().

```
require 'Datavyu_API.rb'
begin
  test = create_mutually_exclusive("test", "col1", "col2")
  setColumn("test", test)
end
```

---

deleteCell()

deleteCell(*cell*)

Removes the specified cell from its column and propagates the changes to the spreadsheet.

Parameter	Type	Description
cell	RCell	Ruby representation of the Datavyu cell to be deleted.

Returns

Undefined.

Example

Removes cells from the "trial" column with "condition" coded as "a".

```
# First get the column from the database
trial = getVariable("trial")

# Now loop through all of the cells in that column, checking if
# they are coded as a left hand.
for trial_cell in trial.cells
    # Is hand coded as "l" for this cell?
    if trial_cell.condition == 'a'
        deleteCell(cell)
    end
end
```

deleteVariable()

deleteVariable(*column*)

Deletes a column from the spreadsheet.

Alias(es): delete_column

Parameter	Type	Description
column	String or RColumn object	Name of the variable that you are adding arguments to.

Returns

Nothing.

Example

The following example removes column 'trials' from the spreadsheet.

```
require 'Datavyu_API.rb'
begin
    deleteVariable('trials')
end
```

getCellFromTime()

getCellFromTime(*col*, *time*)

Identifies the cell that occurs at a given point in time for the specified column, and returns it.

Parameter	Type	Description
col	String or RColumn object	Name or Ruby representation of the column that you are looking for a cell within.
time	Integer	Time (in milliseconds) that you want to identify the cell that happens then.

Returns

Returns the Ruby representation of the cell at the specified point in time. If there is no cell at that point in time, Ruby does not return anything.

Example

The following example identifies the cell that occurs at 100ms in the "trial" column, and assigns it to a RCell object. It then prints out the cell's ordinal, onset, and offset codes for easy location.

```ruby
require 'Datavyu_API.rb'
begin
   trial = getColumn("trial")
   cell = getCellFromTime(trial, 100)

   # Get the ordinal, onset, offset
   # values from the cell, and assign them to
   # string variables, so we can print them out
   ordinal = cell.ordinal.to_s
   onset = cell.onset.to_s
   offset = cell.offset.to_s

   # Print out ordinal, onset, and offset, and their values
   puts "ordinal: #{ordinal}"
   puts "onset: #{onset}ms"
   puts "offset: #{offset}ms"

end
```

getColumn()

getColumn(*name*)

Retrieves a variable from the Datavyu spreadsheet and assigns it to a Ruby object using print_debug().

Parameter	Type	Description
name	String	The name of the Datavyu column you wish to retrieve

Returns

A Ruby object representation of the Datavyu column.

Example

The following example retrieves the Datavyu column "trial" and assigns it to a Ruby variable called trial.

```
require 'Datavyu_API.rb'
begin
  trial = getColumn("trial")
end
```

getColumnList()

getColumnList()

Outputs a list of all the columns in the current spreadsheet.

Parameter	Type	Description
None		

Returns

List of columns.

Example

The following example assigns the list of columns to a Ruby object called, columnList and prints it out using puts.

```
require 'Datavyu_API.rb'
begin
  columnList = getColumnList()
  puts columnList
end
```

load_db()

load_db(*filename*)

Loads a spreadsheet's data directly from the file.

Parameter	Type	Description
filename	String	The full path to the saved Datavyu file.

Returns

- $db: the spreadsheet of the opened project
- $pj: project data of the opened project

Example

The following example loads the test.opf spreadsheet located on the Desktop.

```
require 'Datavyu_API.rb'
begin
    $db,$pj = load_db("~/Desktop/test.opf")
end
```

load_macshapa_db()

load_macshapa_db(*filename*, *write_to_gui*, **ignore_vars*)

Opens an old, closed MacSHAPA spreadsheet file and loads it into the current open spreadsheet.

> **Warning:** load_macshapa_db() only reads in matrix and string columns. It does not yet support predicates, and queries are not imported. In order to be compatible with Datavyu, all times will be converted to milliseconds.

[t]|l|l|L|

Parameter	Type	Description
filename	String	The full path to save the saved MacSHAPA file.
write_to_gui	Boolean	If true, the MacSHAPA file is read into the spreadsheet that is currently open in Datavyu's GUI. If false, the MacSHAPA file is read into the Ruby interface.

Returns

$db, the spreadsheet data and $pj, the project data for that file.

Example

The following example loads the test.opf MacSHAPA file and into ruby variables called $db and $pj.

```
require 'Datavyu_API.rb'
begin
    $db,$pj = load_db("~/Desktop/test.opf", FALSE)
end
```

Example

In this example, the test.opf MacSHAPA file is read into the spreadsheet that is currently open in Datavyu's GUI.

```
require 'Datavyu_API.rb'
begin
    $db, $pj = load_db("~/Desktop/test.opf", TRUE)
end
```

makeDurationBlockRel()

makeDurationBlockRel(*relname*, *var_to_copy*, *binding*, *block_dur*, *skip_blocks*)

Makes a duration-based reliability column. This creates two columns, one containing a cell with a number for that block, and another blank column for the free coding within the block.

[t]|l|L|L|

Parameter	Type	Description
relname	String	The name of the reliability column you are creating
var_to_copy	RColumn object	Name of the column which you are copying, i.e.the existing column that you are creating a reliability column from.
binding	String	Column to bind the copy to.
block_dur	Integer	Length the blocks should be (in seconds)
skip_blocks	Integer	Determines the amount of space that should be left between each coding block. skip_blocks is an Integer. Each skipped block is the length specified by block_dur. If block_dur is 10 seconds, and skip_blocks is 5, then 50 seconds will be left between each coding block.

Returns

Nothing. Columns are automatically written to the spreadsheet.

Example

The following example creates a duration-based reliability column from the "step" column.

```
require 'Datavyu_API.rb'
```

makeReliability()

makeReliability(*relname, column_to_copy, multiple_to_keep, *codes_to_keep*)

Creates a *reliability column* that is a copy of another Datavyu column in the Database. makeReliability() can copy the cells (or a subsection of the cells) and retain codes from the origin column if desired.

[t]|l|L|L|

Parameter	Type	Description
relname	String **or** Ruby column from getColumn()	the name of the new reliability column you will create. The convention is to name it rel_columnName, but you can name it whatever you want to.
column_to_copy	String	The name of the column that we want to create a reliability column *from* (i.e. the existing coded column).
multiple_to_keep	Integer (**optional**)	Number of cells to skip: a value of 2 includes *every other cell* in the new variable; 1 includes every cell, and 0 creates a blank column with no cells
*codes_to_keep	comma-separated strings (**optional**)	Codes you want to copy from the original to new column. These are codes that the reliability coder will not have to code.

Returns

A Ruby object representation of the reliability column within Datavyu.

Example

The following example creates the reliability column "rel_trial" from the primary column "trial", copying every second cell, and retaining the "onset", "trialnum" and "unit" codes, and then writes the new "rel_trial" column back to the spreadsheet.

```
require 'Datavyu_API.rb'
begin
  rel_trial = makeReliability("rel_trial", "trial", 2, "onset", "trialnum", "unit")
  setVariable("rel_trial", rel_trial)
end
```

See also:

- *Check Inter-rater Reliability to Improve Data Accuracy*

printAllNested()

printAllNested()

Parameter	Type	Description
file	String	Path to the Datavyu spreadsheet file whose data you want to print

Returns

[stuff it returns]

Example

[example]

printCellCodes()

printCellCodes(*cell*)

Prints out the values for every code in a specified cell.

Parameter	Type	Description
cell	RCell object	The cell whose codes you are printing out.

Returns

An object listing all of the codes in a given cell.

Example

The following example uses puts to print out the codes for the first cell in the "trial" column, accessed using printCellCodes().

```ruby
require 'Datavyu_API.rb'
begin
   trial = getColumn("trial")
   cell = trial.cell[0]

   puts printCellCodes(cell)
end
```

print_codes()

print_codes(*cell, file, codes*)

Writes a cell's codes to a file, separated by tab (\t).

Parameter	Type	Description
cell	RCell object	Cell whose codes you want to write to a file.
file	String	Path to the file that you wish to write the cell's codes to.

Returns

Nothing. Writes results to the specified file.

Example

The following example uses print_codes() to write the coded values for the cells in the "trial" variable to a file called "trial_codes.txt", located on the Desktop.

```ruby
require 'Datavyu_API.rb'

begin
  # Defines the location of the file that we're going to be outputting
  # the column data to - the file name is baby_codes.txt
  # and is located on the Desktop.
  out_file = File.expand_path("~/Desktop/baby_codes.txt")

  # Creates the file, and assigns write permissions 'w'
  out = File.new(out_file,'w')

  # Retrieves the "BabyLocation" column from the spreadsheet
  baby = getColumn("BabyLocation")

  # Define which codes we want to print out
  codes_to_print = ["ordinal", "onset", "offset", "arg01"]

  # Iterate through every cell in the BabyLocation column to
  # print its coded values.
  for cell in baby.cells
    # Write the ordinal, onset, offset, and code01 codes to the baby_codes.txt file,
    # which is accessed by the variable called out
    print_codes(cell, out, codes_to_print)

    # Write a newline to the file so that the values for each cell
    # will be in their own row
    out.write("\n")
  end
end
```

save_db()

save_db(*filename*)

Saves the current $db and $pj variables to a file. If the filename ends with .csv, save_db() saves the data as a .csv file. Otherwise, it saves it as .opf.

Parameter	Type	Description
filename	String or Ruby object	The full path to save the Datavyu file to

Returns

Nothing.

Example

The following example saves the current spreadsheet open in the GUI to a file called test.opf that is located on the Desktop.

```
require 'Datavyu_API.rb'
begin
  save_db("~/Desktop/test.opf")
end
```

setColumn()

setColumn(*name*, *var*)

setColumn() writes columns to the spreadsheet: for columns that already exist, setColumn() replaces the data in the spreadsheet with the version updated using the script. For instance, if you were to retrieve the "trial" column from a spreadsheet and then make some changes, you would use setColumn() to write those changes to the spreadsheet, replacing the old data with your new data.

If the column does not already exist in the spreadsheet (for instance, if you create a new column using makeNewColumn()), setColumn() will instead create a new column in the spreadsheet.

Parameter	Type	Description
name	String (**optional**)	Name of the *column* that you are inserting
column	RColumn object (**required**)	Ruby container of the column that you are inserting into the spreadsheet (modified output of createNewColumn() or getColumn())

Important: You must specify a value for the column parameter. If you are also passing a value for the name parameter, the order of arguments **must** be name followed by column.

Returns

None

Example

The following example retrieves the Datavyu column "trial" and assigns it to a Ruby variable called trial. After some modifications to the trial object, it writes those changes back to the spreadsheet using setColumn().

```
require 'Datavyu_API.rb'
begin
  trial = getColumn("trial")
  <some modifications to trial>
  setColumn("trial", trial)
end
```

smoothColumn()

smoothColumn(*colname, tol=33*)

Tweaks cell onsets so that there is a maximum of tol milliseconds between each cell. If two cells are less than tol apart, it moves on to the next pair of cells; if there is a larger gap than tol, the second cell's *onset* is set to the first cell's *offset*.

Parameter	Type	Description
colname	String	Name of the column that you wish to modify.
tol	Integer	The tolerance you are willing to accept between a cell's *offset* and the next cell's *onset*. By default, this is 33ms.

Returns

Nothing. In addition, smoothColumn() automatically writes its changes back to the spreadsheet, so you do not need to write the changes using setColumn().

Example

The following example checks the "trial" column's cells to ensure that a maximum of 50ms between a cell's *offset* and the subsequent cell's *onset*.

```
require 'Datavyu_API.rb'
begin
  smoothColumn("trial", 50)
end
```

transfer_columns

transfer_columns(*db1, db2, remove, *varnames*)

Transfers columns between spreadsheets. If db1 or db2 is set to the empty string "", then that spreadsheet is spreadsheet open in the GUI.

Thus, if you want to transfer a column into the GUI, set db2 to "", and specify the origin spreadsheet file as db1. If you want to transfer a column from the GUI into a file, set db1 to "", and set db2 to that file's path.

> **Warning:** Setting remove to TRUE will **DELETE THE COLUMNS YOU ARE TRANSFERRING FROM DB1**.

Parameter	Type	Description
db1	String	The full path to save the first Datavyu file. Set to "" to use the spreadsheet that's currently open.
db2	String	The full path to save the second Datavyu file. Set to "" to use the spreadsheet that's currently open.
remove	Boolean	If TRUE, Datavyu will delete the columns from db1 as they are copied to db2. FALSE leaves the columns intact.
varnames	list of strings	List of the names of the columns that you wish to copy from db1 to db1. You must specify at least one column name.

Returns

Nothing. Saves files in place or modifies the GUI.

Example

The following example transfers the column "idchange" from test.opf to the GUI and leaves test.opf intact and unmodified.

```ruby
# Transfer column(s) from one opf to another.
# Set sourceFile and destinationFile to the full path;
# Leave blank to indicate currently open file.
# This will work for as many columns as you like.

require 'Datavyu_API.rb'

class RVariable

    # Define a way to construct a new cell by cloning a given cell.
    # Not for general purpose use as vocabulary can become corrupt if used improperly
    def make_new_cell2(cell)
        c = RCell.new
        c.onset = cell.onset
        c.offset = cell.offset
        c.ordinal = cell.ordinal
        c.set_args(cell.argvals,@arglist)
        c.parent = @name
        @cells << c
        return c
    end
end

def transferMyVariable(db1,db2,delete,*varnames)
    # If varnames was specified as a hash, flatten it to an array
    varnames.flatten!

    $debug=true

    # Display args when debugging
    if $debug
        puts "="*20
        puts "#{__method__} called with following args:"
        puts db1,db2,delete,varnames
        puts "="*20
    end

    # Handle degenerate case of same source and destination
    if db1==db2
        puts "Warning: source and destination are identical.  No changes made."
        return nil
    end

    # Set the source database, loading from file if necessary.
    # Raises file not found error and returns nil if source database does not exist.
    db1path = ""
    begin
        if db1!=""
```

```ruby
                db1path = File.expand_path(db1)
                if !File.readable?(db1path)
                        raise "Error! File not readable : #{db1}"
                end
                puts "Loading source database from file : #{db1path}" if $debug
                from_db,from_proj = loadDB(db1path)
        else
                from_db,from_proj = $db,$proj
        end
rescue StandardError => e
        puts e.message
        puts e.backtrace
        return nil
end

# Set the destination database, loading from file if necessary.
# Raises file not found error and returns nil if destination database does not exist.
db2path = ""
begin
        if db2!=""
                db2path = File.expand_path(db2)
                if !File.writable?(db2path)
                        raise "Error! File not writable : #{db2}"
                end
                puts "Loading destination database from file : #{db2path}" if $debug
                to_db,to_proj = loadDB(db2path)
                #$db,$proj = loadDB(db2path)
        else
                to_db,to_proj = $db,$proj
        end
rescue StandardError => e
        puts e.message
        puts e.backtrace
        return nil
end

# Set working database to source database to prepare for reading
$db,$pj = from_db,from_proj

# Construct a hash to store columns and cells we are transferring
puts "Fetching columns..." if $debug
begin
        col_map = Hash.new
        cell_map = Hash.new
        for col in varnames
                c = getColumn(col.to_s)
                if c.nil?
                        puts "Warning: column #{c} not found! Skipping..."
                        next
                end
                col_map[col] = c
                cell_map[col] = c.cells
```

```ruby
                    puts "Read column : #{col.to_s}" if $debug
                end
        end

        # Set working database to destination database to prepare for writing
        $db,$pj = to_db,to_proj

        # Go through the hashmaps and reconstruct the columns
        begin
            for key in col_map.keys
                    col = col_map[key]
                    cells = cell_map[key]
                    arglist = col.arglist

                    # Construct a new variable and add all associated cells
                    newvar = createVariable(key.to_s,arglist)
                    for c in cells
                            newvar.make_new_cell2(c)
                    end
                    setVariable(key.to_s,newvar)
                    if $debug
                        puts "Wrote column : #{key.to_s} with #{newvar.cells.length} cells"
                    end
            end
        rescue StandardError => e
                puts "Failed trying to write column #{col}"
                puts e.message
                puts e.backtrace
                return nil
        end

        # Finally, save the database to file if applicable
        saveDB(db2path) if db2path!=""
end

begin
  $debug=false
  sourceFile="/Users/datavyutester/Desktop/FileName1.opf"
  destinationFile="/Users/datavyutester/Desktop/FileName2.opf"
  columnsToTransfer = ["reltrial"]
  transferMyVariable(sourceFile, destinationFile, false, columnsToTransfer)
end
```

See also:
The Glossary

Version 1.3.6

Classes and Class Methods

CTable Class
class CTable

69

Represent a contingency table / confusion matrix for a single code.

classmethod add(*pri_value*, *rel_value*)

Increment the table value at the given combination by one. See compute_kappa() for automatically computing kappa scores.

Parameter	Type	Description
pri_value	String	Value for primary coder.
rel_value	String	Value for reliability coder.

Returns

None.

classmethod ef(*idx*)

Return the expected frequency of agreement by chance for the given index.

Parameter	Type	Description
idx	Integer	Index of code (starting at zero).

classmethod efs()

Return the sum of the expected frequency of agreement by chance for all indices in table.

classmethod kappa()

Compute kappa using table values.

classmethod total()

Return the sum of all elements in table.

classmethod to_s()

Return formateed string to display the table.

RCell Class

class RCell

The Ruby container for Datavyu cells.

classmethod change_code(*code*, *val*)

Changes the value of a code in a cell.

Parameter	Type	Description
arg	String **or** Ruby column from get_column()	Name of the code that you are updating.
val	String, Integer, etc.	Value to change the code to.

Returns

None.

Example

The following example sets the "trial" column's cell at position 0's *onset* to 1000ms and then writes the change back to the spreadsheet using set_column().

```ruby
require 'Datavyu_API.rb'
begin
  trial = getColumn("trial")
  trial.cells[0].change_code("onset", 1000)
  setColumn("trial", trial)
end
```

classmethod is_within(*outer_cell*)

Determines if a cell is temporally encased by the outer_cell.

[t]|l|L|L|

Parameter	Type	Description
outer_cell		The cell that is going to be checked to see if it temporally encases the study cell.

Returns

Boolean

Example

Compare the first cell of the "trial" and "id" columns to see if the first cell of "trial" is temporally enclosed by the first cell of "id". If it is, print out "Yes, it is temporally enclosed", otherwise, print "No, it is not temporally enclosed."

```ruby
require 'Datavyu_API.rb'
begin
  trial = getColumn("trial")
  id = getColumn("id")
  if trial.cells[0].is_within(id.cells[0])
    puts "Yes, it is temporally enclosed."
  else
    puts "No, it is not temporally enclosed."
  end
end
```

classmethod contains(*inner_cell*)

Determines if a cell temporally encases the inner_cell.

[t]|l|L|L|

Parameter	Type	Description
inner_cell		The cell that is going to be checked to see if it is temporally encased by the study cell.

Returns

Boolean

Example

Compare the first cell of the "trial" and "id" columns to see if the first cell of "trial" is temporally enclosed by the first cell of "id". If it is, print out "Yes, it is temporally encloses the cell", otherwise, print "No, it is does not temporally enclose the cell."

```
require 'Datavyu_API.rb'
begin
    trial = getColumn("trial")
    id = getColumn("id")
    if id.cells[0].is_within(trial.cells[0])
        puts "Yes, it is temporally encloses the cell."
    else
        puts "No, it is does not temporally enclose the cell."
    end
end
```

classmethod print_all(*p)

Dumps all of the codes in a *cell* to a *string*.

Parameter	Type	Description
p **optional**	String	The separator between codes. Defaults to tab (t)

Returns

String of the codes, starting with ordinal, onset, and offset, followed by the codes.

Example

The following example prints all of the "trial" column's first cell's codes using print.

```
require 'Datavyu_API.rb'
begin
    trial = getColumn("trial")
    print trial.cells[0].print_all()
end
```

RColumn Class
class RColumn

The Ruby container for Datavyu columns.

classmethod make_new_cell()

Creates a new blank cell at the end of the column's cell array.

Argument	Type	Description
None		

Returns

Reference to the cell that was just created. Modify the cell using this reference.

Example

The following example creates a new cell at the end of the "trial" column's cell array and assigns its reference to the variable newcell. It then changes newcell's *onset* to 1000ms using change_code() and writes the change back to the spreadsheet using set_column().

```
trial = get_column("trial")
newcell = trial.make_new_cell()
newcell.change_code("onset", 1000)
set_column("trial", trial)
```

classmethod change_code_name(*old_name*, *new_name*)

Renames a code.

[t]|l|L|L|		
Argument	Type	Description
old_name	String	Current name of the code
new_name	String	New name for the code, which will replace old_name

Returns

Nothing.

Example

The following example renames the "trial" column's *bad_code_name* code to *awesome_code_name* and then writes the changes back to the Datavyu spreadsheet:

```
require 'Datavyu_API.rb'
begin
  trial = get_column("trial")
  trial.change_code_name("bad_code_name", "awesome_code_name")
  set_column("trial", trial)
end
```

classmethod add_code(*name*)

Adds a code to a column.

[t]|l|L|L|		
Argument	Type	Description
name	String	The name of the code you are adding to the column

Returns

Nothing.

Example

The following example adds the *unit* code to the "trial" column and then writes the changes back to the spreadsheet using set_column().

```
require 'Datavyu_API.rb'
begin
  trial = get_column("trial")
  trial.add_code("unit")
  set_column(trial)
end
```

classmethod remove_code(*name*)

Deletes a code from a column.

Argument	Type	Description
name	String	The name of the code you wish to delete from the column

Returns

Nothing.

Example

The following example removes the *unit* code from the "trial" column and then writes the changes back to the spreadsheet using set_column().

```ruby
require 'Datavyu_API.rb'
begin
  trial = get_column("trial")
  trial.remove_code("unit")
  set_column(trial)
end
```

Freestanding Methods

add_codes_to_column()

add_codes_to_column(*column, *codes*)

Alias(es): *addCodesToColumn, add_args_to_var*

Add new codes to a column.

Parameter	Type	Description
column	String or RColumn object	Name of the variable that you are adding arguments to.
*codes	List	List of arguments to add to column. Parameters with an * indicate lists.

Returns

Ruby representation of the column.

Example

The following example adds "condition_abc", "response_xyz", and "score_123" to the "test" column, and writes it back to the spreadsheet using set_column().

```ruby
require 'Datavyu_API.rb'
begin
  test = add_codes_to_column("test", "condition_abc", "response_xyz", "score_123")
  setColumn(test)
end
```

check_datavyu_version()

check_datavyu_version(*minVersion, maxVersion*)

Checks whether the current version of Datavyu meets the minimum and maximum requirements specified in the parameters.

Requires Datavyu 1.3.5 or greater.

Parameter	Type	Description
minVersion	String	Minimum version to check against; e.g. 'v:1.3.5'
maxVersion	String (optional)	Maximum version to check against.

Returns

True if the current Datavyu version meets the specified requirements; false otherwise.

Example

Raise an error message unless the script is run on Datavyu version 1.3.5 or greater.

```
raise "This script will not work on the current version of Datavyu" if not check_datavyu_version(
↪'v:1.3.5')
```

check_reliability()

check_reliability(*main_col, rel_col, match_arg, time_tolerance, dump_file*)

Compares two Datavyu columns to check for reliability errors and accuracy.

Parameter	Type	Description
main_col	String **or** Ruby variable from getColumn()	The primary column that rel_col will be compared against
rel_col	String **or** Ruby variable from getColumn()	The reliability column to compare to main_col
match_arg	String	The argument used to match the reliability cells to the primary cells. Must be a unique identifier between the cells.
time_tolerance	Integer	Amount of slack permitted, in milliseconds, between the two onset and offsets before it will be considered an error. Set to 0 to tolerate no difference.
dump_file	String path or Ruby File object (**optional**)	The full string path to dump the reliability output to. This can be used for multi-file dumps or just to keep a log. You can also give it a Ruby File object if a file is already started.

Returns

Console and file output.

Example

The following example checks the reliability column "rel_trial" against the primary column "trial", linking the two on their "trialnum" code, with a 100ms *onset* and *offset* difference tolerated.

```
check_reliability("trial", "rel_trial", "trialnum", 100)
```

Example

The following example performs the same operation as the previous example, but also writes the output to ~/Desktop/Relcheck.txt, a text file.

```
check_reliability("trial", "rel_trial", "trialnum", 100, "~/Desktop/Relcheck.txt")
```

See also:
Check Inter-rater Reliability to Improve Data Accuracy

check_valid_codes()

check_valid_codes(*column, dump_file, *arg_code_pairs*)

Alias(es): *checkValidCodes*

Checks that all coded values in Datavyu conform to a the list of valid codes for that column.

Argument	Type	Description
column	String	The Datavyu column that to check
dump_file	String, or Ruby File object	Full path of the file to dump output to. Use "" to write to the console. You can also specify a Ruby File object.
*arg_code_pairs	Key-value pairs	List of code names and valid values, in the format "code_name", ["valid1", "valid2"], "code_name2", ["valid3", "valid4"], etc.

Returns

Nothing. Generated messages are output to console and/or file.

Example

The following example checks the validity of the codes for the "trial" Datavyu column:

```
check_valid_codes("trial", "", "hand", ["l","r","b","n"], "turn", ["l","r"],
    "unit", ["1","2","3"])
```

See also:
Check for Coding Errors

check_valid_codes2()

check_valid_codes2(*data, dump_file, *arg_code_pairs*)

Advanced version of *check_valid_codes()*, available in Datavyu version 1.3.5 and higher.

Can check codes using patterns and can operate over multiple columns. Backwards-compatible with *check_valid_codes()* so this function should be able to replace calls to *check_valid_codes()*.

Argument	Type	Description
data	String, RVariable, or Hash	When this parameter is a String or a column object from *getVariable()*, the function operates on codes within this column. If the parameter is a Hash (associative array), the function ignores the arg_code_pairs arguments and uses data from this Hash. The Hash must be structured as a nested mapping from columns (either as Strings or RVariables) to Hashes. These nested hashes must be mappings from code names (as Strings) to valid code values (as either lists (Arrays) or patterns (Regexp)).
dump_file	String, or Ruby File object	Path of the file to dump output to. Use empty String (i.e. "") to write to the console. You can also specify a Ruby File object (e.g. from *File.open()*).
*arg_code_pairs	Key-value pairs	List of code names and valid values, in the format "code_name", ["valid1", "valid2"], "code_name2", ["valid3", "valid4"], etc. This is ignored if first argument is a Hash.

Returns

Nothing. Generated messages are output to console and/or file.

Example

The following example checks three columns for valid code values. Before the call to the function, a nested mapping is created for each column. The inner map is a mapping from the names of codes to their valid values.

```
## Params
date_format = /\A\d{2}\/\d{2}\/\d{4}\Z/ # dates must be formatted: ##/##/####
# Associative mapping from column names to mappings from code names to valid values
map = {
  'id' => {
    'testdate' => date_format,
    'idnum' => /\A\d{3}\Z/, # id number must be exactly 3 digits
    'gender' => ['m', 'f', '.'], # gender can be one of 3 values
    'birthdate' => date_format
  },
  'condition' => {
    'cond_ab' => ['a', 'b'] # condition can be either 'a' or 'b'
  },
  'trial' => {
    'trialnum' => /\A\d+\Z/, # trial number must be one or more digits
    'result_xyz' => ['x', 'y', 'z'] # result must be one of 3 values
  }
}

## Body
check_valid_codes2(map, '~/Desktop/check.txt')
```

See also:
Check for Coding Errors

combine_columns()

combine_columns(*name*, **columnNames*)

Combines two columns together, creating a new column. create_mutually_exclusive() combines the two source columns' cells together so that the new column includes all of the arguments from both source columns. It also includes a combination of the two columns' cells as well as a new cell for each overlap.

[t]|l|L|L|

Parameter	Type	Description
name	String	The name of the new column you wish to create
columnNames	List of strings	The names of the source column that will be combined

Returns

The new Ruby representation of the column. You must write the changes back to the spreadsheet if you want to save them.

Example

77

The following example creates a column called "test" from two existing columns, "col1" and "col2", and then writes the changes back to the spreadsheet using set_column().

```ruby
require 'Datavyu_API.rb'
begin
  test = combine_columns("test", "col1", "col2")
  setColumn("test", test)
end
```

compute_kappa()

compute_kappa(*pri_col, rel_col, *codes*)

Computes Cohen's kappa for a primary and reliability column. Cells between the two columns are matched by their onset time. Computes a contingency table and kappa score for each specified code.

[t]|l|L|L|

Parameter	Type	Description
pri_col	String or RColumn object	Name (or column object) of primary coder's column.
rel_col	String or RColumn object	Name (or column object) of reliability coder's column.
*codes	List	Names of codes to compute scores for.

Returns

Hashes (associative arrays) for kappa values and CTable, in that order. Keys are names of the codes. Values are Numeric, and :class`Ctable`, respectively.

create_mutually_exclusive()

create_mutually_exclusive(*name, col1name, col2name*)

Combines two columns together, creating a new column. create_mutually_exclusive() combines the two source columns' cells together so that the new column includes all of the arguments from both source columns. It also includes a combination of the two columns' cells as well as a new cell for each overlap.

[t]|l|L|L|

Parameter	Type	Description
name	String	The name of the new column you wish to create.
col1name	String	The name of the first source column to combine.
col2name	String	The name of the second source column to combine.

Returns

The new Ruby representation of the column. You must write the changes back to the spreadsheet if you want to save them.

Example

The following example creates a column called "test" from two existing columns, "col1" and "col2", and then writes the changes back to the spreadsheet using setColumn().

```ruby
require 'Datavyu_API.rb'
begin
  test = create_mutually_exclusive("test", "col1", "col2")
  setColumn("test", test)
end
```

delete_cell()

delete_cell(*cell*)

    Removes the specified cell from its column and propagates the changes to the spreadsheet.

Parameter	Type	Description
cell	RCell	Ruby representation of the Datavyu cell to be deleted.

Returns

Undefined.

Example

Removes cells from the "trial" column with "condition" coded as "a".

```
# First get the column from the database
trial = get_column("trial")

# Now loop through all of the cells in that column, checking if
# they are coded as a left hand.
for trial_cell in trial.cells
   # Is hand coded as "l" for this cell?
   if trial_cell.condition == 'a'
      delete_cell(cell)
   end
end
```

delete_variable()

delete_variable(*column*)

    Deletes a column from the spreadsheet.

    Alias(es): delete_column

Parameter	Type	Description
column	String or RColumn object	Name of the variable that you are adding arguments to.

Returns

Nothing.

Example

The following example removes column 'trials' from the spreadsheet.

```
require 'Datavyu_API.rb'
begin
   delete_variable('trials')
end
```

get_cell_from_time()

get_cell_from_time(*col*, *time*)

Identifies the cell that occurs at a given point in time for the specified column, and returns it.

Parameter	Type	Description
col	String or RColumn object	Name or Ruby representation of the column that you are looking for a cell within.
time	Integer	Time (in milliseconds) that you want to identify the cell that happens then.

Returns

Returns the Ruby representation of the cell at the specified point in time. If there is no cell at that point in time, Ruby does not return anything.

Example

The following example identifies the cell that occurs at 100ms in the "trial" column, and assigns it to a RCell object. It then prints out the cell's ordinal, onset, and offset codes for easy location.

```ruby
require 'Datavyu_API.rb'
begin
   trial = getColumn("trial")
   cell = get_cell_from_time(trial, 100)

   # Get the ordinal, onset, offset
   # values from the cell, and assign them to
   # string variables, so we can print them out
   ordinal = cell.ordinal.to_s
   onset = cell.onset.to_s
   offset = cell.offset.to_s

   # Print out ordinal, onset, and offset, and their values
   puts "ordinal: #{ordinal}"
   puts "onset: #{onset}ms"
   puts "offset: #{offset}ms"

end
```

get_column()

get_column(*name*)

Retrieves a variable from the Datavyu spreadsheet and assigns it to a Ruby object using print_debug().

Parameter	Type	Description
name	String	The name of the Datavyu column you wish to retrieve

Returns

A Ruby object representation of the Datavyu column.

Example

The following example retrieves the Datavyu column "trial" and assigns it to a Ruby variable called trial.

```
require 'Datavyu_API.rb'
begin
    trial = get_column("trial")
end
```

get_column_list()

get_column_list()

Outputs a list of all the columns in the current spreadsheet.

Parameter	Type	Description
None		

Returns

List of columns.

Example

The following example assigns the list of columns to a Ruby object called, columnList and prints it out using puts.

```
require 'Datavyu_API.rb'
begin
    columnList = get_column_list()
    puts columnList
end
```

get_datavyu_version()

get_datavyu_version()

Return the current Datavyu version as a String (e.g. 'v:1.3.5')

Requires Datavyu 1.3.5 or greater.

Returns

String.

load_db()

load_db(*filename*)

Loads a spreadsheet's data directly from the file.

Parameter	Type	Description
filename	String	The full path to the saved Datavyu file.

Returns

- $db: the spreadsheet of the opened project
- $pj: project data of the opened project

Example

The following example loads the test.opf spreadsheet located on the Desktop.

```
require 'Datavyu_API.rb'
begin
    $db,$pj = load_db("~/Desktop/test.opf")
end
```

load_macshapa_db()

load_macshapa_db(*filename*, *write_to_gui*, *ignore_vars)

Opens an old, closed MacSHAPA spreadsheet file and loads it into the current open spreadsheet.

> **Warning:** load_macshapa_db() only reads in matrix and string columns. It does not yet support predicates, and queries are not imported. In order to be compatible with Datavyu, all times will be converted to milliseconds.

Parameter	Type	Description
filename	String	The full path to save the saved MacSHAPA file.
write_to_gui	Boolean	If true, the MacSHAPA file is read into the spreadsheet that is currently open in Datavyu's GUI. If false, the MacSHAPA file is read into the Ruby interface.

Returns

$db, the spreadsheet data and $pj, the project data for that file.

Example

The following example loads the test.opf MacSHAPA file and into ruby variables called $db and $pj.

```
require 'Datavyu_API.rb'
begin
    $db,$pj = load_db("~/Desktop/test.opf", FALSE)
end
```

Example

In this example, the test.opf MacSHAPA file is read into the spreadsheet that is currently open in Datavyu's GUI.

```
require 'Datavyu_API.rb'
begin
    $db, $pj = load_db("~/Desktop/test.opf", TRUE)
end
```

make_duration_block_rel()

make_duration_block_rel(*relname*, *var_to_copy*, *binding*, *block_dur*, *skip_blocks*)

Makes a duration-based reliability column. This creates two columns, one containing a cell with a number for that block, and another blank column for the free coding within the block.

Parameter	Type	Description
relname	String	The name of the reliability column you are creating
var_to_copy	RColumn object	Name of the column which you are copying, i.e. the existing column that you are creating a reliability column from.
binding	String	Column to bind the copy to.
block_dur	Integer	Length the blocks should be (in seconds)
skip_blocks	Integer	Determines the amount of space that should be left between each coding block. skip_blocks is an Integer. Each skipped block is the length specified by block_dur. If block_dur is 10 seconds, and skip_blocks is 5, then 50 seconds will be left between each coding block.

Returns

Nothing. Columns are automatically written to the spreadsheet.

Example

The following example creates a duration-based reliability column from the "step" column.

```
require 'Datavyu_API.rb'
```

make_reliability()

make_reliability(*relname*, *column_to_copy*, *multiple_to_keep*, **codes_to_keep*)

Creates a *reliability column* that is a copy of another Datavyu column in the Database. make_reliability() can copy the cells (or a subsection of the cells) and retain codes from the origin column if desired.

Parameter	Type	Description
relname	String **or** Ruby column from getColumn()	the name of the new reliability column you will create. The convention is to name it rel_columnName, but you can name it whatever you want to.
column_to_copy	String	The name of the column that we want to create a reliability column *from* (i.e. the existing coded column).
multiple_to_keep	Integer (**optional**)	Number of cells to skip: a value of 2 includes *every other cell* in the new variable; 1 includes every cell, and 0 creates a blank column with no cells
*codes_to_keep	comma-separated strings (**optional**)	Codes you want to copy from the original to new column. These are codes that the reliability coder will not have to code.

Returns

A Ruby object representation of the reliability column within Datavyu.

Example

The following example creates the reliability column "rel_trial" from the primary column "trial", copying every second cell, and retaining the "onset", "trialnum" and "unit" codes, and then writes the new "rel_trial" column back to the spreadsheet.

```
require 'Datavyu_API.rb'
begin
    rel_trial = make_reliability("rel_trial", "trial", 2, "onset", "trialnum", "unit")
    setVariable("rel_trial", rel_trial)
end
```

See also:

- *Check Inter-rater Reliability to Improve Data Accuracy*

merge_columns()

merge_columns(*column_names*)

Combines multiple columns together and returns a new column containing all the codes from the given columns (plus an ordinal number code for each column).

Behaves similar to *create_mutually_exclusive*, however, identical results are not guaranteed.

Parameter	Type	Description
name	String	The name of the new column you wish to create.
columns	List	Columns to combine. List can contain either the names of the columns as Strings or the RColumn object representing the column.

Returns

New RColumn object on success. Nil value on failed merge.

Example

The following example creates a column called "merged" from three source columns: gesture_mom, gesture_child, gesture_dad.

```
merged_column = merge_columns('merged', 'gesture_mom', 'gesture_child', 'gesture_
↪dad')
set_column(merged_column) # save to spreadsheet.
```

new_column()

new_column(*name*, **codes*)

Creates a new blank column with the specified name and codes.

Parameter	Type	Description
name	String	The name of the new column
codes	comma separated Strings	Codes that the new column will contain.

Note: new_column() creates the onset, offset, and ordinal codes by default. You do not need to specify them in the codes.

Returns

Ruby object representation of the new column in Datavyu.

Example

The following example creates a new Datavyu column called "trial" with the codes "trialnum" and "unit", and assigns them to an RColumn object called trial. It then adds a new cell to trial using new_cell() and writes the changes back to the Datavyu spreadsheet using set_column().

```
require 'Datavyu_API.rb'
begin
   trial = new_column('trial', 'trialnum', 'unit')
   trial.new_cell()
   set_column(trial)
end
```

print_cell_codes()

print_cell_codes(*cell*)

Prints out the values for every code in a specified cell.

Parameter	Type	Description
cell	RCell object	The cell whose codes you are printing out.

Returns

An object listing all of the codes in a given cell.

Example

The following example uses puts to print out the codes for the first cell in the "trial" column, accessed using print_cell_codes().

```
require 'Datavyu_API.rb'
begin
   trial = getColumn("trial")
   cell = trial.cell[0]

   puts print_cell_codes(cell)
end
```

print_codes()

print_codes(*cell*, *file*, *codes*)

Writes a cell's codes to a file, separated by tab (\t).

Parameter	Type	Description
cell	RCell object	Cell whose codes you want to write to a file.
file	String	Path to the file that you wish to write the cell's codes to.
codes	Array of Strings	Codes whose coded values you wish to print.

Returns

Nothing. Writes results to the specified file.

Example

The following example uses print_codes() to write the coded values for the cells in the "trial" variable to a file called "trial_codes.txt", located on the computer Desktop.

```ruby
require 'Datavyu_API.rb'

begin
    # Defines the location of the file that we're going to be outputting
    # the column data to - the file name is baby_codes.txt
    # and is located on the Desktop.
    out_file = File.expand_path("~/Desktop/baby_codes.txt")

    # Creates the file, and assigns write permissions 'w'
    out = File.new(out_file,'w')

    # Retrieves the "BabyLocation" column from the spreadsheet
    baby = getColumn("BabyLocation")

    # Define which codes we want to print out
    codes_to_print = ["ordinal", "onset", "offset", "arg01"]

    # Iterate through every cell in the BabyLocation column to
    # print its coded values.
    for cell in baby.cells
        # Write the ordinal, onset, offset, and code01 codes to the baby_codes.txt file,
        # which is accessed by the variable called out
        print_codes(cell, out, codes_to_print)

        # Write a newline to the file so that the values for each cell
        # will be in their own row
        out.write("\n")
    end
end
```

save_db()

save_db(*filename*)

Saves the current $db and $pj variables to a file. If the filename ends with .csv, save_db() saves the data as a .csv file. Otherwise, it saves it as .opf.

Parameter	Type	Description
filename	String or Ruby object	The full path to save the Datavyu file to

Returns

Nothing.

Example

The following example saves the current spreadsheet open in the GUI to a file called test.opf that is located on the Desktop.

```
require 'Datavyu_API.rb'
begin
  save_db("~/Desktop/test.opf")
end
```

set_column()

set_column(*name*, *var*)

set_column() writes columns to the spreadsheet. For columns that already exist, set_column() replaces the data in the spreadsheet with the version updated using the script. For instance, if you were to retrieve the "trial" column from a spreadsheet and then make some changes, you would use set_column() to write those changes to the spreadsheet, replacing the old data with your new data.

If the column does not already exist in the spreadsheet (for instance, if you create a new column using makeNewColumn()), set_column() will instead create a new column in the spreadsheet.

[t]|l|L|L|

Parameter	Type	Description
name	String (**optional**)	Name of the *column* that you are inserting
column	RColumn object (**required**)	Ruby container of the column that you are inserting into the spreadsheet (modified output of createNewColumn() or getColumn())

Important: You must specify a value for the column parameter. If you are also passing a value for the name parameter, the order of arguments **must** be name followed by column.

Returns

None

Example

The following example retrieves the Datavyu column "trial" and assigns it to a Ruby variable called trial. After some modifications to the trial object, it writes those changes back to the spreadsheet using set_column().

```
require 'Datavyu_API.rb'
begin
  trial = getColumn("trial")
  <some modifications to trial>
  set_column("trial", trial)
end
```

smooth_column()

smooth_column(*colname*, *tol=33*)

Tweaks cell onsets so that there is a maximum of tol milliseconds between each cell. If two cells are less than tol apart, it moves on to the next pair of cells; if there is a larger gap than tol, the second cell's *onset* is set to the first cell's *offset*.

[t]|l|L|L|

Parameter	Type	Description
colname	String	Name of the column that you wish to modify.

tol	Integer	The tolerance you are willing to accept between a cell's *offset* and the next cell's *onset*. By default, this is 33ms.

Returns

Nothing. In addition, smooth_column() automatically writes its changes back to the spreadsheet, so you do not need to write the changes using setColumn().

Example

The following example checks the "trial" column's cells to ensure that a maximum of 50ms between a cell's *offset* and the subsequent cell's *onset*.

```
require 'Datavyu_API.rb'
begin
  smooth_column("trial", 50)
end
```

transfer_columns()

transfer_columns(*db1*, *db2*, *remove*, **varnames*)

Transfers columns between spreadsheets. Replacing db1 or db2 with the empty string "", will refer instead to the currently open spreadsheet in Datavyu.

Thus, if you want to transfer a column into the GUI, set db2 to '', and specify the origin spreadsheet file as db1. If you want to transfer a column from the GUI into a file, set db1 to '', and set db2 to that file's path.

> **Warning:** Setting remove to TRUE will **DELETE THE COLUMNS YOU ARE TRANSFERRING FROM DB1**.

Parameter	Type	Description
db1	String	The full path to save the first Datavyu file. Set to '' to use the spreadsheet that's currently open.
db2	String	The full path to save the second Datavyu file. Set to '' to use the spreadsheet that's currently open.
remove	Boolean	If TRUE, Datavyu will delete the columns from db1 as they are copied to db2. FALSE leaves the columns intact.
varnames	list of strings	List of the names of the columns that you wish to copy from db1 to db1. You must specify at least one column name.

Returns

Nothing. Saves files in place or modifies the GUI.

Example

The following example transfers the column "idchange" from test.opf to the GUI and leaves test.opf intact and unmodified.

```
require 'Datavyu_API.rb'
begin
  sourceFile='/Users/datavyutester/Desktop/FileName1.opf'
  destinationFile='/Users/datavyutester/Desktop/FileName2.opf'
  columnsToTransfer = ['trial_rel', 'condition_rel']
  transferMyVariable(sourceFile, destinationFile, false, columnsToTransfer)
end
```

See also:
The Glossary
See also:
The Glossary

Chapter 3

Frequently Asked Questions

3.1 General

3.1.1 What is Datavyu used for?

Datavyu is a video coding and data visualization tool for collecting behavioral data from video.

3.1.2 Can I use Datavyu to analyze data?

Datavyu is a data coding tool, not a statistical analysis tool. You can export data from Datavyu in a variety of forms depending on your analysis needs. See the *Export Data from Datavyu* and *Use Scripts to Export Data from Datavyu* tutorials for more information.

3.1.3 What video formats does Datavyu support?

Datavyu supports all video formats that Quicktime or VLC can play, including .mp4, .asf, .wmv, .avi, .flv, .mov, .mpf, .ogg, .mpg, .nsc, .wav, and .dts. Please test video playback on each of your computers in your work environment by downloading Datavyu, and playing video data you plan to code.

We highlight suggest using the Quicktime plugin because the VLC plugin has known timing problems which could lead to inconsistent coding.

3.1.4 Can I code sources that are not videos?

Yes, we mostly support video data but other sources can be analyzed in Datavyu as well.

3.1.5 Is there somewhere I can share or store my spreadsheets and videos?

Yes indeed! Databrary is a web-based repository for open sharing and preservation of video data and associated metadata. The project website has more information about the initiative and a guide to help you get started.

3.1.6 Do I have to pay for Datavyu?

No! Datavyu is a completely free and open source program. If you would like to contribute to the development of Datavyu, see: the source on GitHub.

3.1.7 What is Datavyu's citation?

Datavyu Team (2014). Datavyu: A Video Coding Tool. Databrary Project, New York University. URL http://datavyu.org.

3.2 Technical Requirements

3.2.1 Which operating system is Datavyu available for?

Datavyu builds are available for both Windows and Mac OS X. You can download either version from the Datavyu Downloads page.

3.2.2 Do I need any additional software to run Datavyu?

Datavyu requires Java, and either Quicktime or VLC. See *Requirements* for more details.

3.2.3 What hardware do I need to code data sources in Datavyu?

Datavyu makes extensive use of a the keypad for controlling video playback. If your keyboard does not have a keypad (for example, if you're working on a laptop), you will need to acquire an external keypad.

Beyond that, there are no specific hardware requirements.

3.2.4 Do I need to be connected to the internet when using Datavyu?

No! Datavyu runs entirely on your computer. If you are connected to the internet, however, Datavyu will check to see if there is a new version of Datavyu that you can download. See *Keep Datavyu Up-to-Date* for more information.

3.3 Support

3.3.1 I've never coded video data before. How should I start?

We are writing a Best Practices Guide to provide detailed tips and advice to help you start coding behavioral data from video.

3.3.2 Where can I learn to use Datavyu?

The *Datavyu user guide* provides comprehensive documentation of Datavyu's interfaces and capabilities, as well as tutorials to guide you through common tasks.

3.3.3 I'm not sure how to best code my data? Are there guidelines for coding?

The Best Practices Guide will provide in-depth instructions and suggestions for coding your behavioral data. It is coming soon!

3.3.4 What is the best way to prevent data loss?

We suggest that users pull all of their spreadsheets to the desktop before coding them. After coding, users can ship their spreadsheets back to their folders or hard drive.

3.3.5 How do I update Datavyu?

When you first start up Datavyu while connected to the internet, Datavyu automatically checks for new versions and prompts you to update if there is a new version. See: *Keep Datavyu Up-to-Date* for more information.

3.3.6 Where can I ask questions about issues I'm having with Datavyu?

Post your questions to the Datavyu support forum and Datavyu maintainers and users will help you find the answers you're looking for.

3.3.7 I have found a bug in Datavyu! How can I report it?

If you find a bug, you can post to the Datavyu support forum or email the Datavyu team, Datavyu Support.

3.4 API Scripts

3.4.1 What is a script?

A script is a collection of code that performs actions on a file or spreadsheet. You could write a script to add a column, add codes to a column, duplicate and move data around, or export data to a convenient file format for analysis with SPSS, for example. See the *Ruby API guide* for more information.

3.4.2 I'm not a programmer - where can I learn to write scripts?

The *Ruby API guide* provides an in-depth introduction to scripting with Datavyu's Ruby API, tutorials that guide you through common scripting tasks and detailed method reference for each API component, to help you learn how these components work together.

If you're completely unfamiliar with programming, going through the Ruby language's Learn Ruby in Twenty Minutes tutorial may be helpful. Ruby is an easy-to-read programming language which was designed to be intuitive and easy to learn, so gaining the basics should be feasible.

3.4.3 What can I use scripts for?

You can write scripts to import, manipulate, and export data. The *API Tutorials* describe common tasks you can write scripts to accomplish.

3.5 MacSHAPA and OpenSHAPA

3.5.1 I have existing MacSHAPA files - can I convert my them to Datavyu?

Yes! You can import a MacSHAPA file into Datavyu through a simple script. See *Convert MacSHAPA Files to Work in Datavyu* for details.

3.5.2 I have existing OpenSHAPA files - can I convert them to Datavyu?

Yes! You can open your OpenSHAPA files in Datavyu like you would any other Datavyu file.

3.5.3 Will my MacSHAPA queries work in Datayvyu?

Unfortunately, MacSHAPA queries do not work on the new Datavyu format. You will need to write new scripts for your Datavyu spreadsheets.

3.5.4 Will my OpenSHAPA scripts run in Datavyu?

Yes! You can easily convert an OpenSHAPA script to work in Datavyu by adding require 'Datavyu_API.rb' to the top of your OpenSHAPA script file and removing the OpenSHAPA API code that precedes the script. For more information see the *Convert an OpenSHAPA Script to the Datavyu Format* tutorial.

Chapter 4

Walkthrough Videos

We have produced several videos that help illustrate Datavyu's strengths and guide users through common tasks:

4.1 Tutorial Videos

4.1.1 Datavyu Components & Playback

Learn about Datavyu's media player and video controller. Discover the power of Datavyu's fingertip playback control. For more information see *Getting Started* and *Controller Overview*.

4.1.2 Time-Lock Events Codes to Video: Cells & Coding Spreadsheet

Learn how Datavyu time-locks video events to the coding spreadsheet. Dive into how a coding column and a "cell" within a coding column work. See examples of time-locking events to video. For more information see *Spreadsheet Overview* and *Configure Columns and Codes*.

4.1.3 Time-Lock Events Codes to Video: Coding Timestamps

Learn how to use Datavyu's fingertip control functions to time-lock video to the coding spreadsheet. Explore three ways of inserting timestamps with appropriate use cases. For more information see :doc:'/guide/tutorials/add-cells'.

4.2 Short Walkthrough Videos

4.2.1 Modifying Columns & Font Size

See how to create, rename, and move columns. Learn how to increase and decrease font size on cell codes by "zooming."

4.2.2 Modifying Codes with the Code Editor

See how the Code Editor can be used to create new columns and codes, rename codes, and change code order.

4.2.3 Spreadsheet Temporal Alignment

Watch how cells can be viewed relative to their duration across columns using temporal alignment in the coding spreadsheet.

4.2.4   Modifying Columns with a Script

See an example of how scripts can be used to modify coding spreadsheets. Learn about scrips with *Ruby API documentation*

Chapter 5

Coding Example

5.1   Watch an Expert User Code

Use the provided coding example and video to practice coding.

5.2   Watch an Expert User Transcribe

Watch a transcriber use Quick Key Mode to tag utterance timestamps and use Highlight and Focus to insert transcripts into those newly created cells.

Chapter 6

Best Practices for Coding Behavioral Data from Video

6.1 Watch a Presentation on Video Coding and Best Practices

Databrary and Datavyu are hosting regional workshops around the U.S. to share research tools and resources to the science community. You can watch the regional workshop presentations and view the slides on Databrary.

6.1.1 Overview of Coding Process

Welcome to the Best Practices Guide. These guidelines are intended as general suggestions for how to code behavioral data from video. The guidelines will help you to make the most of Datavyu, but the general principles are applicable for coding with any software tool or even for coding with paper and pencil.

Datavyu is agnostic about what researchers code and how they code it. This makes the software very powerful and flexible, but it puts the responsibility of designing the spreadsheet and coding criteria on the user. For a beginner setting out to code behavioral data for the first time, or a more experienced coder who is new to Datavyu, figuring out where to start can be daunting. This guide will help you to get started and will provide a framework for thinking about coding behavioral data from video.

If you have questions or comments about behavioral coding, please go to the Datavyu Support Forum. Other researchers may have run into the same problems, posted similar requests, or have offered similar suggestions for improving the coding process. Similarly, other researchers may benefit from hearing your questions and comments.

While learning about best practices in behavioral coding, you may find it useful to reference the *Datavyu User Guide* to learn more about Datavyu features and various aspects of the Datavyu spreadsheet and video controller.

Video Coding as a Series of Filters

Your video files are your raw material. Video can't capture everything that happens in a session, but with a well-designed recording arrangement, video can capture the essential behaviors of interest. The data that you actually analyze (e.g., with statistics) are not the raw video files. Instead, the data that you analyze are derived from a smaller subset of information—categorical codes, durations computed from onset and offset times, straight transcripts of speech, informal comments, and so on. Therefore, it is important that your video recording arrangement allows your coders to see the behaviors of interest, that your codes reflect the information you intend to capture, and that the data are in a format that permits you to run the analyses you want to do. As Bakeman (2000) put it: "Occasionally investigators speak of videotapes as data, but this seems a misnomer. Videotapes...are raw material, not data. Data...are the product of measurement; videotapes are no more data than a hunk of marble is sculpture" (p. 144).

Think of the video coding process as a series of filters. Your recording arrangement is the first filter. Your participants emit behavior. Although video captures much of the richness and complexity of behavior, your cameras cannot capture everything. Some of the behavior and some of the context pass through the initial filter and that is what you capture on video. But some of what the participants do and some of the physical and social context is immediately lost. Your video codes are the second filter. You will score only a small subset of behaviors visible and audible on video. The rest of the behaviors are unused and unexamined. What you choose to code, depends on your theoretical perspective. Your analyses are the third filter. Time-tagged video coding (as enabled by Datavyu) provides information about the timing relations among coded behaviors. However, most researchers analyze only the frequencies and durations of behaviors they code and the timing relations (the order of events; what happens first and last; time lags between events; etc.) are lost.

Poor choices in terms of your video recording arrangement and coding scheme will taint the entire process. Good choices will highlight the questions you wish to address by focusing the camera, coding, and analysis filters on the behaviors of interest.

4 Steps of Video Coding

Coding behavioral data is a multi-step, iterative process of planning, testing, revising, and refining. You should expect to revise many aspects of your coding plan as the data present you with new information and surprises. Even highly experienced coders should not expect to plan everything ahead of time because new tasks, procedures, populations, and research questions affect participants' behaviors. Indeed, one of the great joys of coding behavior is that you will always discover new things. Thus, you will likely need to rethink your plan a few times. Fortunately, getting started and making revisions and additions are easy to do in Datavyu.

The overall process involves four steps, which are described in detail below. These four steps will help you to avoid wasting time. We urge you not to collect hours of video before verifying that you can see the behaviors of interest. Do not code hours of data before verifying that your coding scheme is reliable and that you can export the data in a format that works for your statistical and graphical analyses. Otherwise, you may spend valuable time collecting and coding data that you can't use.

4 Steps of Video Coding

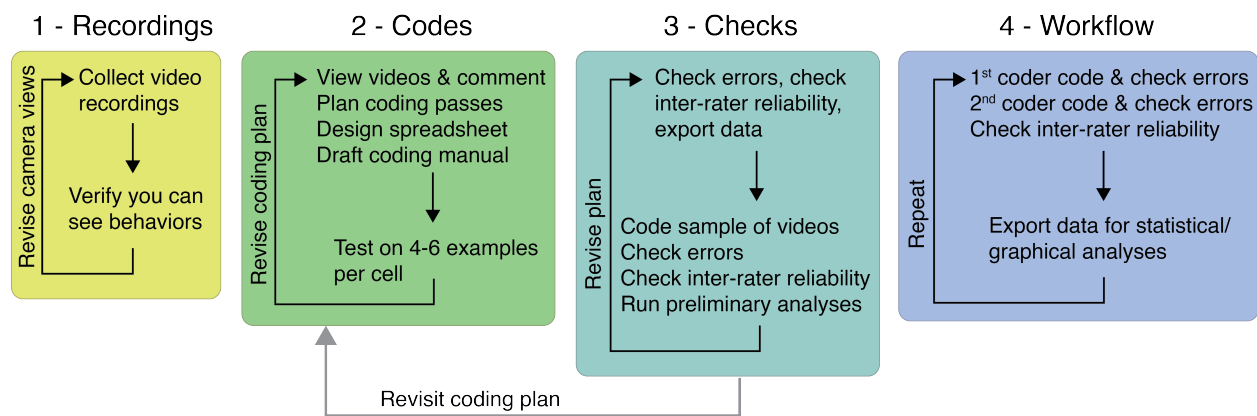Figure1: The process begins at Step 1 with a procedure for ensuring that your recordings capture the behaviors of interest; the process ends at Step 4 with an exported file of variables that can be used for statistical and graphical analyses. Each step involves multiple iterations (denoted by the looping arrows). Ideally, earlier steps are completed appropriately before implementing later steps.

6.1.2 Step 1: Recordings—Verify That You Can See the Behaviors of Interest

Overview

Record a few videos with your intended study procedure. Verify that your video recording arrangement allows your coders to see the critical behaviors of interest. If the behaviors are not visible, revise the camera views. Once verified, begin collecting data in earnest.

What It Means to "See" Behaviors of Interest Remember, the video camera is a filter. If the behaviors are not on camera, you cannot see them and you cannot code them; they might as well have never happened. Datavyu allows you to view your videos at various speeds. Exploit this feature to determine whether the behaviors of interest are adequately visible for coding.

On Camera Camera views make all the difference. With well-designed camera views, the behaviors of interest are visible and large on the video frame. It can be advantageous to collect data from multiple camera views: different behaviors may be visible from different angles, and it is often helpful to have multiple perspectives on a single behavior. Datavyu can synchronize views from different video files or you can use commercial software to merge and synchronize your camera views onto a single video frame. Often, the critical behavior of interest is obscured on one camera view; multiple camera views ensures that you will always be able to see the behavior.

Viewing Speed Visibility varies depending on the viewing speed. For many behaviors, you will want to view the videos at speeds slower than real time. Datavyu provides easy, fingertip control over the viewing speed with the shuttle keys—allowing you to slow down, speed up, or pause the video with a tap of a finger. You can also view the videos frame by frame using fingertip control over the jog keys.

Rules of Thumb Use HD video for a crisper, higher resolution image. But, be aware of data storage requirements. Most people don't need the highest setting on their camera, which results in huge, unmanageable files. Large files also require more powerful computers for playback. The "lowest" HD setting is a lot easier to store and for most purposes looks great. Thus, consider the trade-off between higher resolution videos and storage/processing requirements. Your camera lenses and apertures will also affect whether the image is blurry, distorted, dark, or over-exposed.

In general, it is easier to code things you can see rather than things that you hear.

Use visual cues, not sound cues to demarcate sections of video. You can only hear sound while playing the video in real time and it is difficult to determine when a sound begins or ends with frame precision. Use visual contrast, not sound, to demarcate important sections of your recording session (e.g., new conditions, new trials). Abrupt changes in contrast are easier to see than subtle visual changes, especially at faster than real-time speeds (and you are likely to want to fast forward to different sections of your video). Flipping the room lights on and off provides an easily implemented and visible demarcation of sections of the session. Holding a bright card in front of a camera (with condition name or trial number) provides another easily implemented and visible demarcation of sections of the session.

Use multiple camera views to capture both small body parts/small movements and the whole body/scene. Small body parts, small movements, facial expressions, and eye movements are difficult to see when they are small in the video frame.

Part of planning your recording arrangement involves thinking about your recording context. For example, white skin is difficult to see against a light background and dark skin is difficult to see against a dark background. So, create more visual contrast by making your background a bright or saturated color or dressing the child in a bright or saturated color (we use bright blue carpets and mats because infants' hands and feet are clearly visible against them).

Video Example This video displays how easy it is to record videos that miss the behaviors of interest. The experimenter was interested in understanding how parents teach children to open containers. You can see by the video example that the single camera view is completely blocked by the parent.

Revise or add more camera views until you can thoroughly see your behaviors of interest.

How To Determine Whether Your Recording Arrangement Is Optimal   After you've collected data from a few sessions, open the video files in Datavyu and view representative portions of the videos at various speeds, including real time, speeds faster and slower than real time (1x, 1/2x, 2x with the shuttle keys, and frame-by-frame using the jog keys), and while moving backward through the video at various speeds (-1/2x, -1x, -2x, jogging backward, etc).

Are the critical behaviors on camera?

Do the zoom and camera angle make it easy to see the behaviors of interest?

Can you see the frame when a behavior of interest starts and the frame when it ends or is it too blurry?

If you cannot see and hear the behaviors of interest, you will need to revise your recording arrangement or rethink your goals for coding.

6.1.3   Step 2: Codes—Design a Formal Coding Scheme

Overview

To design a formal coding scheme, you will need to plan your coding passes, set up a template Datavyu spreadsheet, and draft a coding manual that defines your codes. Test your coding scheme on a few examples for each cell in your study design (e.g., each age and condition). As your draft criteria fail or new behaviors emerge, revise your coding scheme (remember that revisions are to be expected). Consider the types of analyses that you would like to run as you plan your codes and spreadsheet. Be sure that you can export your data in a suitable form for your analyses and that your coding scheme is not too detailed or too vague for your statistical and graphical analyses. You may have beautifully coded and reliable data, but if you cannot export it in a way that allows for analyses, you have nothing.

Coding Theory and Practice

Useful Definitions   You may want to reference *Datavyu's User Guide* to gain deeper understanding of various key terms.

A *code* tags a section of video with an identifier. Codes for outcome measures typically reflect the expression or non-expression of particular behaviors or traits. Codes can also represent participant information, conditions, tasks, predictors, and independent variables. In Datavyu, codes are represented by a cell in the coding spreadsheet or by a variable within the cell. When coders "code" video files, they insert cells and type letters into prompts for each variable within the cell.

A *coding manual* describes and documents what coders should do (and what previous coders did do) while scoring the videos. It formalizes the coding decisions by defining what each code represents, and the criteria for coders' decisions. This information is valuable for researchers who may analyze or revisit their data months or years after its collection, for setting conventions within and across labs, and for sharing and repurposing data. You likely want a separate coding manual for each study. You can use an existing manual as a template to set up a new coding manual.

A *coding pass* reflects a complete scoring of a video file for one variable or set of variables. In Datavyu, a pass is generally a column of cells with codes.

A *spreadsheet* organizes and stores your codes for a particular video file. In Datavyu, the spreadsheets are the Datavyu files. Each spreadsheet is automatically linked with its corresponding video file. In Datavyu, your codes are in cells and your cells are in columns. Regardless of coding software, you should expect to develop your coding manual and set up your template coding spreadsheet in tandem. This is an iterative process and you will likely need to make changes or want to add new coding passes and/or codes down the line.

A *comment* is a note by a coder. Comments can be completely informal and used effectively to get an idea of what behaviors of interest are on the videos. Comments can be more formalized (by adding the coder's name and date of comment) and used in a more serious way to locate excerpts, highlight problems/discrepancies, or explain a coding decision.

The *ordinal* is the number of the cell in the sequence of cells in a column. Ordinals are an important way to keep track of a sequence of cells or the identity of a cell when time is not useful.

In Datavyu, *onset* and *offset* times are the two times that accompany each cell. Typically, the onset time marks the beginning of an event and the offset time marks the end of the event. Sometimes events are continuous (e.g., when baby looks left, the look to the right ends; when baby looks away, the look to the left ends). Sometimes events are isolated (e.g., after trial #1 ends, there are several seconds or minutes before trial #2 begins). Sometimes only the onset or the order of events is important; in that case, you can code using point cells, where there is only one time associated with each cell (onset and offset are the same number). Sometimes onsets and offsets are arbitrary (maybe you want to assess a behavior every two minutes or randomly sample 10 minutes of behavior from each hour); Datavyu scripts make time sampling easy. Datavyu has special keys for entering onsets/offsets for continuous events, isolated events, and point cells. Sometimes you will want to use onset or offset times as a way to link cells across columns. Note that the notion of onset and offset is only a convention. In Datavyu, it is possible for the "onset" time to be later than the "offset" time, if for example, you want one time to represent the start of event A and the other time to represent the start of event B and sometimes B precedes A. In these cases, Datavyu will display the cell with a red line (don't worry, the red line is only a tag; it does not mean that you made a mistake unless your code does not allow offset times to precede onset times).

In Datavyu, a *script* is a routine written in the Ruby programming language that allows you to manipulate the data in your spreadsheet, add/delete codes from your spreadsheet, insert or delete cells in your spreadsheet, insert or delete columns in your spreadsheet, conduct analyses on your codes, import data into the spreadsheet, and export spreadsheet data in whatever format you desire. Scripts can operate on a single spreadsheet or on all of the spreadsheets in a file (e.g., 100s of spreadsheets simultaneously).

Coding Criteria and Types of Codes Behavioral codes lie along a continuum. Implicit (sometimes called "subjective") codes are at one end of the continuum and explicit (sometimes called "objective") codes are at the other end of the continuum. The difference between the two types of codes is whether the behavioral criteria for codes are implicit or explicit. Implicit codes do not require the observer to see particular behaviors; explicit codes do require this. As illustrated in the following 2 x 2 table, the benefits of one are the failure of the other. Implicit criteria allow coders to determine the code based on their own judgments of what behavior is being expressed; coders can take individual differences between participants into account. Explicit criteria force coders to determine the code based on whether a particular behavior was expressed; individual differences between participants and particulars of the situation must be ignored. With an implicit code for "falling," for example, coders use their own judgment to decide whether the infant lost balance in the particular instance. With an explicit code for "falling," coders must use explicit criteria such as whether the infant's hands or bottom touched the floor, whether the transition from upright occurred within a particular time frame, or whether an experimenter or parent grasped the infant's body. With an implicit code for "negative affect," coders use their own judgment to decide whether a child feels distress or anger. With an explicit code, coders must use explicit criteria such as whether the child's brows were knit, lip was jutted, mouth was in a square shape, or crying/tears were expressed.

Type	Implicit Coding Criteria	Explicit Coding Criteria
Pros	Reflect individual differences in the manner of expressing the target behavior	Know how behavior was expressed in each instance and individual
Cons	Do not know how behavior was expressed in each instance and individual	Ignores individual differences in manner of expressing the target behavior

Implicit and explicit codes can be equally reliable (in terms of inter-rater reliability and consistency of participants' responses) and equally valid (meaning that the codes reflect the behaviors you intend to measure). The benefit of an explicit code is that you will know exactly what coders scored (e.g. baby stopped for at least 0.5s at the edge of the obstacle with feet or hands touching the obstacle, etc.).

In some cases, implicit codes are your best bet and can assure you that an explicit code was sufficiently exhaustive. For example, in a study asking whether infants defer to mothers' advice about walking down slopes, we worried that mothers' delivery of encouragement or discouragement might have been influenced by the severity of the slope of an incline; perhaps mothers did not encourage as enthusiastically on steep slopes as they did on shallow ones or did not discourage as enthusiastically on shallow slopes as they did on steep ones. Thus, we asked blind coders to judge whether the slope was shallow, steep, or intermediate and whether mothers were providing an encouraging or discouraging message based solely on mothers' behaviors.

With this implicit code, coders judged type of message nearly perfectly, but judged degree of slope exactly at chance. We thus satisfied ourselves that the explicit codes were sufficiently exhaustive.

Implicit and explicit codes can be of any granularity. Datavyu can provide detailed frame-by-frame coding (in milliseconds) and global approximate coding (region of video, ordinal only) or both. It's entirely up to you. Implicit and explicit codes can refer to durations (involving onset and offset times) and to categories (did behavior occur yes/no; which of several behaviors occurred; what is the ranking of the behavior).

Non-behavioral codes can also be scored in Datavyu. For example, you can type in or import information about participant demographics, the observational setting, various conditions and independent variables, and so on. Non-video data can be imported into Datavyu if you use our code and Ruby API. You can import your own data into Datavyu and use it to identify interesting sections of video or you can use the video to identify interesting sections of other synchronized data streams.

Plan Coding Passes

Before Designing Your Codes   Before planning your formal coding criteria, get an overview of your videos. Watch a few representative segments of video (4-6 videos from each "cell" in your research design) in real time. Watching bits of video from several participants or sessions will save you from basing your coding scheme on behaviors that are not representative of your whole sample.

In Datavyu, you can create a "comment" column where you can jot down your ideas as you watch the videos. Your off-the-cuff observations will be tagged to the approximate location of the event in the video that prompted your thoughts. You needn't even pause the video to do this.

When you feel that you've seen enough, start planning your formal coding scheme. You can refer back to your comments and the corresponding portion of video using the "find" key on the Data Viewer Controller.

Start Simple   Behavior is rich and complex, so it might be tempting to try to code everything at once. Don't do it! Instead, start simple. In Datavyu, you can use columns to capture information about the participant(s) on the video and to delineate important sections of the session and to reflect your study design. You can also use columns to capture the behaviors of interest.

Starting simple is especially important for researchers new to behavioral video coding and users new to Datavyu. Please start simple!

Why Code in Passes?   Coding in passes (scoring one set of measures all the way through a video file) is faster, more efficient, and less tiring for coders than coding multiple passes simultaneously (e.g., watching a trial to score it for one set of measures then watching it again to score it for a second set of measures and so on). Coding in passes minimizes the need to watch the same short bits of video repeatedly to score multiple behaviors. In Datavyu, you can code your variables in any order that you like. However, if you adopt the recommended practice of coding in passes, you will code a set of measures all the way down one column in your spreadsheet before coding another column. Datavyu, however, will allow you to code across columns and to code the same segment of video repeatedly, and has shortcut keys to do so.

You may wish to use your first coding pass to delineate important sections of the session. Perhaps the session begins with some introductory procedures (interview, questionnaire, set-up, etc.), is followed by the target procedure, and then concludes with clean-up procedures. Or perhaps your recording session involves 3 studies or one study with several conditions. In these cases, your first column might reflect the overall temporal structure of the session.

For your first content-loaded coding pass, focus on the behavior(s) most important to your study question (e.g., did the baby say the correct word, did the baby go over the edge of the cliff, etc.). In other words, start with the dependent measure that is most important, direct, and quick to answer the primary question of your study (if you could pick only one dependent measure, this is the one to start with). Focusing your first coding pass on the primary outcome measure(s) ensures that you will not waste time coding a sea of variables that you might never analyze. You can code other, secondary behaviors in subsequent passes (e.g., where baby looked while naming the object).

Passes can be nested and interleaved. Conditions are nested within the participant, trials are nested within conditions, and outcome measures are nested within trials. In Datavyu, you do not need to repeat

the more inclusive category for each nested category. Instead, you can tag each row of behavior with the larger inclusive categories when exporting the data using scripts in the script library.

Natural behaviors are interleaved and overlapping. Some behaviors are ongoing while other behaviors are stopping and starting (e.g. while child is talking, child touches and then stops touching a toy, etc.). Datavyu does not require mutually exclusive codes (e.g. talk without touch, talk with touch, no talking without touch, no talking with touch, etc.). This is a good thing because three types of behaviors are too much for a coder to translate into mutually exclusive categories by themselves (e.g. talk without touch and without look, talk without touch and with look, talk with touch and without look, etc.). You can code behaviors in different passes to capture the interleaving and overlapping. This is much easier than trying to deal with all the combinations of possible events. Exporting interleaved codes can be challenging so you should be comfortable exporting simpler codes before tackling interleaved behaviors. After you are comfortable with exporting simple and nested codes, visit the script library to find scripts for carving interleaved behaviors into mutually exclusive categories for export.

Consider inter-rater reliability as a separate coding pass for each of the coding passes scored by the primary coders. In Datavyu, you can do primary coding and reliability in the same spreadsheet by adding a column for each reliability pass. To ensure that the reliability coder does not inadvertently cheat, you can hide the column coded by the primary coder using the "hide column" feature.

Rule of Thumb: Minimize Pain, Maximize Gain Coding is a repetitive activity: Coders look for particular behaviors and score them over and over across different participants and sessions for hours at a time. So, minimize the requirements on coders' attention and short-term memory. Design your codes so that coders' visual attention is directed toward a coherent set of behaviors that occur at the same relative places in the video frame. For example, if the coder is scoring the person's manual actions, they can easily attend to the objects touched at the same time. Indeed, they cannot determine whether a reach occurred without also noting the target of the reach. Do not ask coders to divide their attention between two regions of the screen simultaneously. It is very difficult to attend to a person's face, for example, while simultaneously attending to the rest of the body; this would divide the coders' attention.

Do not over code. You can always go back and add detail (and Datavyu's scripts will facilitate this process). So, start simple with the behaviors you are most interested in. Do not agonize over frame accuracy if you do not care about the exact durations of an event. Just code what you want to analyze.

A related rule of thumb: If you have already seen it, you may as well code it. That is, if the coder already knows something of interest without doing additional work, looking at additional video frames, or giving the problem additional thought, then the coder may as well code that behavior. For example, if the coders are scoring grasping actions, they also know what object is being grasped without the coders having to think about it or look at any additional video frames; so they may as well code the target object. If the coders are scoring the first frame when a child starts to walk and the last frame in the walking bout, the coders know "for free" which foot the child used to begin walking and which foot the child used to end the bout. However, the coders do not know without additional coding the timing of the other steps within the walking bout.

Reciprocally, do not design codes that require a mental struggle to decide whether something has occurred or not. The more a coder can't decide whether a behavior occurred or which behavior occurred, the more tiring and grueling is the coding process, and the less likely you are to get good, clean data.

Minimize "back-tracking" through the video. Coding is least taxing if the coders can move linearly through an entire video, stopping only to identify the frames to mark onsets and offsets and fill in variable codes within a cell. Thus, the coder might need to wiggle within a few frames to find the target frames, but you do not want your coders to have to backtrack through the same segments of video repeatedly; they should not need to view the same several seconds or minutes of video repeatedly to score variables.

Reduce short-term memory load. Make the prompts for each code transparent and accessible. When possible, do not require your coders to remember the letters for codes (what you set in your coding manual). In Datavyu, you can reduce the memory load by prompting the codes in the code name (e.g., <touch_t_m_b_o> might prompt the coder to score whether the child touched a toy, the mother, self, or other object). You can also reduce coders' memory load by turning codes into yes-no options (e.g., <toy-yn> <mother-yn> <body-yn> <other-yn> can be very quickly scored as y, n, n, n by tabbing through the codes if the child touched a toy). Do not require coders to type 0-1 codes. No one can remember whether 0 = yes

and 1 = no or the other way around (or 0 = male and 1 = female, or 0 = left and 1 = right). Use letters instead. Everyone can remember that y = yes and n = no (and m = male and f = female, etc).

Keep in mind that coding requires motor actions: Coders press keys over and over for hours. Thus, exploit the features of Datavyu that minimize strain on coders' eyes, hands, and brains. In general, you want coders to move their hands as little as possible and their eyes as little as possible. Coders should avoid moving their eyes down to the keyboard and avoid moving their hands from place to place on the keyboard. Use letters that are accessible without moving the hand from a resting position. An example hand position would be using the "home keys" because keyboards are designed to allow the greatest access from that position. Avoid using the mouse! Mousing requires coders to move hands and eyes a lot. Another good reason to avoid numerical codes is that in Datavyu, numbers must be typed using the number row at the top of the keyboard because the number pad is reserved for controlling the video and the spreadsheet. So to type a number, the coders have to move their hands to the top of the keyboard.

Minimize keystrokes by making your codes single letters. You can reuse the same letters in Datavyu because each code stands alone (e.g., you can have 10 codes in a column that are all yes-no variables). Never ever use capital letters. This adds a needless extra key press.

Set up the critical components of Datavyu (video images, Data Viewer Controller, spreadsheet) on your computer screen in the way that best fits your personal preference and minimizes the need to move your eyes and hands.

You will likely appreciate the temporal alignment feature of Datavyu that allows coders to immediately see how one code is nested within another or aligns temporally with another. The alignment feature conserves coders' motor and psychological energy by providing them with an immediate visualization of the data they are coding.

Design the Coding Spreadsheet Design the spreadsheet to make coding and exporting efficient and straightforward. Think through each coding pass one by one.

In Datavyu, each column represents a coding pass. Start with a column containing the participant information and a column (or multiple columns) to delineate sections of the session and conditions within studies and trials within conditions. If your study is highly structured with trials, then you can code the smallest experimental unit (the trial) at the same time as you code your primary outcome measure. Focus first on your most important outcome measures. Include variables in the pass that coders can see "for free" without having to shift their attention or think deeply (refer to *Minimize Pain, Maximize Gain section, If you have already seen it, you may as well code it*).

In Datavyu, each cell in a column can contain zero or multiple codes. A cell will contain zero codes when every cell represents the same type of event and you do not want additional information about each event (e.g., the onset time is when the left foot comes onto the floor and the offset time is when the left foot leaves the floor; or the onset time is when the eyes point at the target and the offset time is when the eyes leave the target).

Because cells correspond to certain times, you may want to define onsets as the start of a behavior and offsets as the end of the behavior, but this is not obligatory. If you don't want to analyze time in terms of durations of behaviors, you don't have to stress over onsets and offsets. But if you do want to analyze durations of behaviors, creating criteria for onsets and offsets is important.

Because each cell corresponds to a particular time (from the onset to offset), it usually makes sense for all of the codes within a cell to correspond to the behaviors within that time interval. However, this is not obligatory. Your cell can include information that occurred before the cell began or after the cell ended (e.g., cell onset = when mother selects a toy and cell offset = when mother offers the toy to the child; a variable within the cell reflects whether child accepted the toy, despite the fact that the acceptance or rejection occurred after the offset of the cell, etc.). Build the prompts for each code with the "minimize pain/maximize gain" rule in mind.

Video Example This video displays how easy it is to set up a draft spreadsheet. After the columns and codes are added through the Code Editor, you can view the prompts provided for the coders.

Draft a Coding Manual Write your coding manual with a stranger in mind. When you revisit your manual years later, you will be a "stranger" to the coding criteria. When new coders open your manual, they

will be strangers to the coding criteria. Write it for a stranger. Do not use acronyms or terms known only to members of your lab. Use plain English instead. If you plan to eventually share your videos on Databrary, you may also want to share your Datavyu files and coding manual.

More detailed documentation is better. For example, if your document contains only information like "o = object touch" and "b = body touch," your coders may not be reliable, the codes may not be replicable, and the researcher who writes up the study may not have sufficient detail about the coding rules. More detail will help. For example: "o = object touch. This behavior includes only touching detached objects that the child could hold in one or both hands; at least one finger must be in contact with the object for at least 0.5 s." And "b = body touch. This behavior includes only touches with one or both hands to the other arm, legs, head/face/hair, and torso; at least one finger must be in contact with the body for at least 0.5 s." It is acceptable to have several paragraphs to define a single code!

As part of the definition of a code, you can specify the optimal speed of viewing for a particular coding pass or code. For example, some behaviors are easier to see by jogging frame by frame. Other behaviors are easier to see at 0.5x normal speed or at normal speed. Sections of video (conditions/trials) can usually be coded at 2x normal speed if they are well marked with a high-contrast prompt.

Outline the coding passes in the manual. List which scripts to run and when to run them to insert cells, merge cells, export data, and so on.

Consider your coding manual as a "living document." Even if you do not revise the codes, you are likely to make changes to the coding manual by adding detail or fixing confusing language. Keep a record of who made the changes and what date they were implemented.

Video Example   This video displays one example of a Coding Manual that outlines the different codes for the pass called "trial." It contains detailed definitions for each code so future coders can easily pick up the coding pass. It also contains pictures to accompany the written descriptions.

Test Your Plan   Test your coding scheme by coding representative portions of video for several participants (a few minutes from 4-6 participants per cell of your research design). It is best to test your coding plan on participants you did not use to design your coding plan. You are likely to find that you will need to revise your coding criteria or add/delete codes when you try out your scheme on new participants, a new age group, or for a new condition. This is normal: behavior is rich and complex and happily, and children do different things under different circumstances. However, if the coding feels unduly arduous and grueling, you should simplify the codes to minimize cost to coders' attention. It is usually better to code in multiple passes than to code in one grueling, painful pass. In Datavyu, exploit the features of the software to make every keystroke count and to make every shift in coders' gaze and attention worth their while. For example, if a behavior occurs earlier in the event, you may need to move the variable to an earlier spot in the variable list.

Remember, designing a formal coding scheme is an iterative process. You need to start somewhere, test it out, and then revise.

6.1.4   Step 3: Checks—Check for Careless Errors, Inter-Rater Reliability, and Design the Format to Export Your Data

Overview

Check that the spreadsheets are free of careless coding errors and that inter-rater reliability is acceptable. Test your initial plan on a small but representative subset of the video data (4-6 participants from each cell of your design). At this point, you can get an idea of whether your coders and codes are likely to be reliable and you can satisfy yourself that you can export your data in the format you need for statistical analyses.

Using Scripts in Datavyu   One of the most powerful and flexible features of Datavyu is the scripting function. In Datavyu, a script is a program written in the Ruby programming language that identifies particular values of codes and durations, writes results to a csv text file, manipulates cells or columns within spreadsheets, performs operations on values within cells, imports data into the spreadsheets, and prints data for export. In Datavyu, you should use scripts to check that the spreadsheets are free of coding

errors and that inter-rater reliability is acceptable. Datavyu has a push-button export function, but this is very rudimentary and will only create a text file that has the same information in the same order as the spreadsheet.

Datavyu scripts are very powerful. You can use them to perform operations on a single spreadsheet linked with one video file or on hundreds of spreadsheets linked with hundreds of videos simultaneously. Thus, if you want to change the name of a code or add or delete a code, you don't need to manually open every spreadsheet and perform the operation from Datavyu's code editor. Instead, you can perform these operations over all the spreadsheets in a folder with one button click of a script. If you want to check your file for typos, you do not need to rely on eyeballing the spreadsheet. Instead, you can write a script to locate any typos. If you want to insert cells to check for inter-rater reliability, you do not need to insert each cell manually. Instead, you can write a script to insert all the necessary cells at pre-specified intervals or random intervals to prompt the reliability coder for onset or offset times and codes.

More generally, whenever you need to perform an operation over many cells or many spreadsheet files (e.g., add new columns/passes, add new variables, change name of variables), use a script. The operation will be nearly instantaneous. Check that the spreadsheets are free of careless coding errors and that inter-rater reliability is acceptable. Test your initial plan on a small but representative subset of the video data (4-6 participants from each cell of your design). At this point, you can get an idea of whether your coders and codes are likely to be reliable and you can satisfy yourself that you can export your data in the format you need for statistical analyses.

Check for Careless Errors   Coders make two kinds of errors. One kind of error is a careless error such as a typo or its equivalent. The coder types a letter that is not a legal option; the coder forgets to mark an offset time; the coder mistakenly inserts an extra cell or numbers trials out of sequence. Coders are human and even the most diligent coder will inadvertently make careless errors. These kinds of errors are not serious and can be easily fixed if they are caught before the coder shuts the file and checks inter-rater reliability.

Consistent use of codes is important for your analyses, but Datavyu will allow you to enter any value you choose. Thus, you need a way to check that your nomenclature is consistently applied within and across video files, that cells were inserted correctly, and that each cell has an appropriate onset and offset time. In Datavyu, you can do this by writing a script to check for careless errors. You should ensure that all of the codes are legal values (according to the coding manual). Check that all of the durations are within acceptable limits (typically negative values are impossible). Check that all of the coded values follow basic logic. If the child did not touch an object, then there can be no object code for that cell. If the child's latency to cross the cliff was 0s, then the child could not have avoided going over the cliff or explored before going over the cliff (because a latency of 0 means there was no time to explore and avoidance reflects the total possible trial time). Thus, your error-checking script will identify typos, impossible relations, and out of range values. In Datavyu, you can even determine the out of range values online based on just-coded files with a script using the R-interface.

Video Example   This video displays one way to check for errors (typos, impossible values, etc.) within a spreadsheet.

Check Inter-Rater Reliability   A second kind of error is an error in judgment. One coder thinks that the child touched an object but a second coder does not think that the child touched the object. One coder interprets the child's facial expression as distress but the other coder sees it as neutral. If coders frequently cannot agree about the codes for the same section of video, your coding scheme lacks inter-rater reliability. Inter-rater reliability will be low if there is a problem with the coding criteria (e.g. criteria are ill defined, criteria do not map well onto the behaviors, etc.) or if the coders are not well trained, or both. So, before you commit yourself to a coding scheme, test inter-rater reliability. If it is too low, you may need to revise your coding scheme or retrain your coders. Disagreements among coders are inevitable, even those who are practiced and familiar with the coding scheme. The question is whether the inter-rater reliability is sufficiently high to warrant confidence in the coded data.

How To Test Reliability Formally What level of agreement is sufficient to consider a code to be reliable. The literature has no gold standard, but labs typically have their own gold standards. Generally, for categorical codes, you should use Kappas (which control for the base rate of the behaviors) rather than percent agreement. If you rely solely on percent agreement, then low frequency events will not be counted fairly. For example, if you are coding child affect as positive/negative and children rarely express negative affect (say, only 2-3 times per 100 trials), one coder could score positive for every trial without even looking at the video and you will have 97% agreement. The Kappa statistic takes low frequency events into account. For continuously scored behaviors, you can use the Kappa statistic to check inter-rater reliability frame by frame. For isolated events, you can use a Pearson correlation coefficient. You can estimate Kappas in Datavyu using scripts and the R interface, or you can estimate Kappas using statistical software after you export your data. Regardless, disagreements among coders are serious and must be reported in the write-up of the research.

Rules of Thumb By definition, careless errors will lower inter-rater reliability. Therefore it is important to check for and eliminate careless errors before you test for inter-rater reliability.

Coders will experience "drift," meaning that their coding will change slightly as they become increasingly experienced at looking at particular behaviors. Therefore, it is important to check inter-rater reliability at every point in the study—on initial sessions, in the middle of the study, and on the final sessions.

How much video should the reliability coder view to ensure inter-rater reliability? A good rule of thumb is 25%. But because every child is different, your reliability coder should score 25% of each child's data, rather than 25% of the data. Which 25%? You can check inter-rater reliability at random intervals or regular intervals—whatever is most appropriate for sampling over the dataset. In some cases, particular trials or segments of video are especially important. In these cases, the reliability coder can score a larger percentage of the data—up to 100%.

Spread your best eyes over the entire dataset. For a large amount of data where several people will split the job of coding a particular pass, have your most experienced and knowledgeable coder score reliability so that your "best pair of eyes" is looking at representative data over the entire dataset.

Video Example This video displays one way to check for inter-rater reliability for a single column in a spreadsheet.

Export the Data in a Format Appropriate for Your Analyses From the beginning of the coding process, keep in mind the data you want to analyze and how you want your data formatted for analyses. Your coded data are what you will analyze. So, you should be sure that the way you code your data is compatible with the way that you will analyze it. Think about how you want your data to look when you export it from Datavyu to analyze it elsewhere (e.g., Excel, SPSS).

Variable Types Although we recommend using alphabet letters as codes rather than numbers, most researchers prefer to analyze numeric data rather than strings. Although onset and offset times contain the information you need to understand timing relations, most researchers prefer to analyze durations rather than relative times.

Datavyu's Export File function will export the strings and raw onset/offset times from the spreadsheet. You can convert these data into numeric and durations in your analysis spreadsheet (with simple compute functions in Excel or SPSS, for example). But you can also export the data in these formats using Datavyu scripts.

Spreadsheet Format Most researchers like to analyze their data in square spreadsheet formats. Many researchers organize the data with one row for the smallest unit of analysis (e.g., a trial, a look, a touch, a facial gesture) and many rows for each participant. Some researchers maintain one row per participant and organize tasks or trials across columns. Some researchers aggregate the data (e.g., by averaging over trials) prior to export to maintain one row per participant.

Behavioral data have a naturally nested structure: Trials are nested within conditions; conditions in turn are nested within sessions and participants. Facial expressions are nested within interactions and interactions

are nested in turn within particular situations within the session. Behavioral data are naturally interleaved: One event is ongoing while another event is starting or ending.

Datavyu's temporal alignment feature provides coders with immediate information about the nested and interleaved temporal structure of events. Thus, we recommend that coders delineate the structure of the session by coding cells to represent the larger and smaller nested units. However, the nested and interleaved structure of events cannot be exported automatically without the user specifying with a script how they would like to see the events formatted. Although Datavyu has an automatic Export File function, it will not repeat information down rows of data unless the data are arranged like that in the spreadsheet. There is no need to repeat participant ID or condition labels across every trial, however, because you can request this with a script for exporting the data. Moreover, Datavyu's automatic Export File function will not carve overlapping and interleaved events into mutually exclusive categorical combinations (e.g., of talking, touching, and looking). You will need to do this with an export script.

Video Example   This video displays a user running a script to export data in a specific way. It exports all of the columns of one spreadsheet into an Excel file. Instead of just exporting one cell from a Datavyu spreadsheet, into one cell in Excel, it makes it possible to repeat important information that you want to store in multiple cells. In this example, participant metadata (id, birthdate, testdate) is stored in only one cell in a Datavyu spreadsheet but it is information that gets repeated down multiple rows of data to make it potentially easier to analyze in a statistical program.

Test Your Plan   After coding representative video files from each cell of your research design, test your plan using scripts. Run scripts to ensure that you will catch careless errors before you export your data. Run scripts to ensure that your codes are reliable among multiple coders. And run scripts to verify that you can export your data in the format you need for analyses. If you cannot export your data using only the Export File function, you will need to use scripts to export your data from Datavyu in the format you want using the Ruby scripting language. Push a small amount of data (preliminary data if you prefer) all the way through to the analysis spreadsheet to assure yourself that your data are in the appropriate format for analyses.

6.1.5   Step 4: Workflow—Establish a Workflow and Code Videos

Overview

Now that you have a functioning coding scheme and manual, and have a plan for exporting and analyzing your data, you should establish a workflow and code videos in earnest. Use a template spreadsheet. Keep your video and Datavyu files and scripts organized into folders. Your primary coder should score a file and then check it for errors. The reliability coder goes next, scoring the file and checking it for errors. The two coders check their reliability and make decisions about disagreements. Then export the data into an analysis spreadsheet.

File Organization   Keep your files well organized. You might want an overall study folder that contains smaller folders to hold paperwork, video files, Datavyu files, and so on.

Establish a standard naming convention for each file type and stick to it. For example, Crawler10-01.mp4, Crawler10-01.opf, Crawler13-23.mp4, Crawler13-23.opf, Walker13-08.mp4, and Walker13-08.opf might represent infants from crawling and walking groups at 10 and 13 months of age; the numbers after the dash might represent their participant IDs; the file extension denotes video and Datavyu spreadsheet files. Naming conventions will make your files easier to find by members of your lab and will make your data easier to share with other labs.

If you keep video files in the same folder on the same path as they were when you originally opened them, Datavyu will automatically find them for you when you open the corresponding spreadsheet. Otherwise, you can link them to the spreadsheet manually. If you keep your Datavyu files in the same folder, you can easily run scripts over all of the files in the folder. You can backup your video files onto a hard drive or you can use Databrary as your video file backup prior to sharing with the larger community.

Keep a formal record of who coded what passes on what video files and what date they did it. Do the same for reliability coding, and for whether discrepancies among coders were checked or discussed. You might also assign coding jobs using the same record-keeping system.

Keep your coding manual up to date. If the codes change, make a note of who implemented the change and what date the change was implemented. This will help you if you need to go back to recode portions of video or if you need to distinguish changes in the code in your analyses.

Template Spreadsheet Keep a template spreadsheet in your study folder on your computer. This spreadsheet has the latest version of the codes for each pass, but the columns are blank. When you are ready to code a new participant, duplicate the template and save it with the appropriate file name. You can keep the template up to date by making changes globally to all of the files in a folder using a script.

File Storage Keep your data safe. Back up your video and Datavyu files. You can store your videos and Datavyu files on Databrary to ensure safe and secure storage and backup.

Coding Workflow Create a workflow that suits the operation of your lab. In general, an efficient workflow minimizes errors and maximizes coder's time on task.

First Coder Code and Check Errors You will have a primary coder or set of coders for each pass. The primary coder will code the pass through the entire video file. When finished, before shutting the spreadsheet, the coder will run the check error script for that pass. The coder will correct all of the identified careless errors. Then the coder will note in your formal records that the pass is finished and ready for the reliability coder.

Reliability Coder Code and Check Errors The reliability coder will code a subset of the video file (25% or so) for that pass. In Datavyu, the reliability coder scores data into a new column and the codes from the primary coder's column are "hidden." When finished, before shutting the spreadsheet, the reliability coder will run the check error script for that pass. The coder will correct all of the identified careless errors. Then the coder will note in the records that the reliability pass is finished.

Check Inter-Rater Reliability Now the coders will check their inter-rater reliability. In Datavyu, to check inter-rater reliability, you will run a script that identifies times, codes, or trials where coders disagreed. The rate of disagreements must be reported in the published report of the study. Likely, the final analyses of inter-rater reliability for that pass will be conducted after all of the files are coded. However, checking inter-rater reliability intermittently ensures that none of the coders experience so much drift that the codes become unreliable.

Typically, researchers analyze the data produced by the primary coder and do not analyze the reliability coder's data beyond ensuring inter-rater reliability. However, known errors need not be entered into the primary data analyses conducted on the column of data produced by the primary coder. Instead, if the coders determine after discussing each disagreement that the error was committed by the primary coder, they could swap the codes between the primary and reliability coders (thereby retaining the same inter-rater reliability), noting the swap if desired. In this way, known errors are eliminated from the final analyses on the primary coder's data. Note that if errors are eliminated in this way, you must keep a record of the original disagreement so that you do not inflate your reliability statistics.

Export Data In Datavyu, data can be exported incrementally after each file is coded (using the File Export function or a script), after an entire coding pass is completed across the entire set of video files (using a script to complete export in one button press), and/or after all of the coding passes are completed. Researchers may prefer to analyze one set of variables as they become available rather than waiting until the entire study is completely coded.

Share Your Data in Databrary Video files and Datavyu files can be stored and shared with lab members and collaborators on Databrary as each session is collected. When the researcher is ready (typically, after a paper describing the study has been accepted for publication), the video files and Datavyu files and other metadata (coding manual, etc.) can be shared with authorized researchers in the Databrary developmental and learning science community.

Turn all paper data into electronic files immediately after the session is completed. This will make it easier for you to track participant permissions, keep your files organized, and share your data. Do not share participant contact information on Databrary.