Yate and the P Word (v5.3)

Yes the dreaded **P** word. Programming! If you're going to be writing actions in Yate you will be programming. Some of the things you can do are trivial while ... well, some are not.

This document will talk about the concepts and the things you should watch out for. It will not present a detailed description of every available action statement.

Yate's actions follow Apple's Automator model. As such is the case you do not use a text editor to enter your actions. You create and edit a statement as an entity which has a display and settings. Due to this model it can be difficult to look at an action statement's textual description and relate it back to the statement which generated it. In an action editor you can do a Show Action Help from the context menu or double click on a statement to show its associated description.

The code snippets displayed in this document can be downloaded and installed so that you can view them in *their natural setting* if you so desire. None of the actions require loaded audio files. The actions can be downloaded as a single zip file and installed in Yate. You can get the actions on the resources page in the Documentation section or directly here. Note that the version number on each action snippet is the same as the version of this document. After downloading the snippets, they will be updated by the Action Updater if they are updated.

The code snippets are all marked as Hidden in the Action manager so that they do not appear on access menus. Further each snippet ensures that it is not inadvertently directly run.

The snippet actions all start with the following statements:

```
1: Version #
2: Cancel with failed run context
3:
```

Statement 1 is a Version statement included so that the Action Updater can report updates. Statement 2 is a **Test Run Context** statement used to stop the action from being run. Statement 3 is an empty line.

The above statements are not included in this document. For that reason snippets in this document start at line number 4.

Put some music on, don't bite your nails and read on.

Table of Contents

Chapter 1 - Audio Files and Fields

Primarily you'll be working on audio files. You can write actions which work entirely on database or text files but we'll forget about that for now. The set of audio files available to you are by default the currently selected files in the main window, or when you're batch processing, a single folder's worth of audio files.

Within audio files you'll be manipulating fields. There are a lot of fields available in every file. Things such as Artist, Album Artist, Album, Title and many other metadata items are fields. The previously mentioned items are *standard* fields. In Yate you can also add up to a hundred Custom fields. Fields are easy to manipulate in actions. Here's a complete list of the built in fields:

> AAC Kind, Album, Album Artist, Artist, BPM, Catalog Number, Category, Classification, Composer, Conductor, Content Advisory, Copyright, Copyright/Legal Information Webpage, Description, Disc, Disc Count, Encoded By, Encoder Settings, Encoding Time, Episode, Episode ID, File Type, Genre, Grouping, Identification, Initial Key, Involved People, ISRC, Label, Language, Length, Love, Lyricist, Media Type, Mood, Movement Count, Movement Name, Movement Number, Music CD Identifier, Musician Credits, Network Name, Official Audio File Webpage, Official Audio Source Webpage, Official Internet Radio Station Webpage, Original Album, Original Artist, Original Filename, Original Lyricist, Original Release Time, Owner, Part of a Compilation, Payment Webpage, Play Count, Playlist Delay, Podcast URL, Price Paid, Produced Notice, Publisher, Publisher's Official Webpage, Purchase Date, Radio Station, Release Time, Remember Position, Remixer, Reverb, Season, Seller, Set Subtitle, Show Description, Show Name, Show Work Name, Skip When Shuffling, Sort Album, Sort Album Artist, Sort Artist, Sort Composer, Sort Show Name, Sort Title, Start Time, Station Owner, Stop Time, Tagging Time, Title, Track, Track Count, Video Definition, Video Description, Volume Adjustment, Work Name, Year

As we said, there are a lot of them.

There are additional field types which are are extensible and may contain multiple items:

> Artwork, Comments, Commercial Information Webpage, Equalization, General Encapsulated Object, Lyrics, Official Artist/Performer Webpage, Private Information, Rating, Relative Volume Adjustment, Terms of Use, Unhandled Items, Unique File Identifier, User Defined Text Info, User Defined URL

Along with Artwork, User Defined Text Information items, or UDTIs for short, are commonly used. UDTIs represent the bulk of tags which are not assigned to one of the built in fields.

Back to Table of Contents

Chapter 2 - Data Types

Strings

Strings are the primary data type in Yate. With only one exception all data types are based on strings. Strings can contain any Unicode character and are not in themselves tied to any one particular character encoding. When audio files are read and written, Yate will use the appropriate character encoding.

With very few exceptions all audio file fields are stored as strings.

Note that there is no concept of **nil**. A string can be empty but it always exists. In Yate if you test a field for existence, you're trying to determine if it's empty ... completely empty. A string with a space in it is not empty!

Numbers

Some Yate statements let you interpret a *string* as an integer value. When this is the case the string representation is converted to an integer value by ignoring everything in the string which cannot be interpreted from the start as an integer. If there is no valid integer that can be extracted from the *string*, zero will be assumed. Be aware that there is never any storage of the integer value. It is always stored as a string representation. For example the **Increment** statement will extract an integer from a field or named variable, will add one to the value and will then convert it back to a string for storage.

There are also action statements which support the use of fractional numbers. The same rules apply as for integers.

Yate also supports boolean values ... in fact some Yate fields such as Part of a Compilation are by definition a boolean value. When reading these values an integer value is extracted. Any non zero value is assumed to be true. When setting a boolean value, true is 1 and false is 0. Some statements will also accept **true** and **false** as values. However, it is safe to assume that zero implies false and non zero implies true.

Lists

When working with audio metadata lots of things are lists. Multiple artists in the same field are a list.

In Yate any text string can be a list by associating it with a specific delimiter. A delimiter can be a single character or a string. Various statements in Yate use fixed delimiters while others let you choose the delimiter.

List processing is a valuable tool as often a complex operation can be performed with a few statements as opposed to looping over various statements. As a general rule, the fewer statements executed the faster an action is.

```
Bob Dylan;;;The Band
```

The above is a list with two elements separated by the **;;;** delimiter. (Which is Yate's default multi-value delimiter).

Key-Value Lists

Key-value lists are simply an interpretation of a text string using two delimiters. One to separate keys (names) from values and one to separate successive key-value pairs. Key-value lists can be used to emulate dictionaries or property lists.

```
Name=Bob Dylan\nRole=Vocals,Guitar
```

The above is a key-value list with two entries separated by the newline sequence (\n). Each name is separated from its value by the **=** character. Note that the value field for **Role** is itself a list with two items separated by a comma (,).

Containers

Containers pretty much break the *everything is a text string* rule. Over the years more and more actions processed downloaded data which was often in JSON format. Containers were implemented so that JSON (JavaScript Object Notation) could be represented and manipulated as a well defined typed structure.

Containers are an efficient means of storing complex data in a format which is easily accessed. Containers are the only non text based data type available in Yate. A container is an array or an object (dictionary). Containers can be created from JSON data, as an empty array or object or as a copy of all or part of another container. A container persists until it is overwritten, removed, or action processing terminates.

A container could represent the following JSON notation:

```
[
  {
    "id" : 23,
    "location" : {
      "address" : "123 Front Street",
      "phone #" : "555-5555"
    },
    "moods" : [
      "rock",
      "blues"
    ],
    "name" : "Jack",
    "special" : true
  },
  {
    "description" : "great person",
    "id" : 24,
    "location" : {
      "address" : "21 Main Street",
      "phone #" : "555-2222"
    },
    "moods" : [
      "classical",
      "opera",
      "jazz"
    ],
    "name" : "Jill",
    "special" : null
  }
]
```

A **Create Container from JSON** statement could create the container.

———— Chapter 2 Snippet 1

```
4: Create container 'Test Container' from JSON text '[ .... ]'
```

————

Back to Table of Contents

Chapter 3 - Variables

There are three types of variables in Yate and another *sort of* variable type.

When an action statement refers to a **Track Variable**, it is referring to a field named **Variable 0**, **Variable 1**, **Variable 2**, ... **Variable 15**. These variables are fields like any other, with the exception that their contents are discarded when action processing is terminated and that their values are never saved in the files. When we get to execution modes, you'll see that it is very useful that files have the track variable fields. They enable the *parellel* processing of many things that you can do on audio files. Oops, another P word!.

Track variables are directly referenced by various statements and can be specified as an escape sequence.

———— Chapter 3 Snippet 1

```
4: Increment Variable 5
5: Append "\v1" to the Album field
6: Set Variable 3 to "Jazz"
```

————

Statement 4 directly references track variable 5. Statement 5 will append the contents of track variable 1 to the Album field. Statement 6 sets track variable 3 to **Jazz**.

Another type of variable is a *Named Variable*. Named variables can be named anything you like. Named variables are not associated with any one audio file. They are application wide and also only exist while an action is running. There are many action statements which set the value of a named variable. When you want to read the value of a named variable, you typically use an escape sequence. We'll come back to escape sequences later. As the escape sequence for named variables uses a **>** character as a terminator, it's a good idea to not place **>** characters in the name. If you reference a named variable which has never been assigned, its value will be the empty string. As actions are modular and might eventually be called by others, it is a good idea to initialize your named variables as opposed to assuming they will be empty.

———— Chapter 3 Snippet 2

```
4: Increment named variable 'counter'
5: Append "\<trailer>" to the Album field
6: Set named variable 'Preferred Genre' to "Blues"
```

————

Statement 4 directly references named variable **counter**. Note that named variables are case insensitive. **counter**, **Counter** and **COUNTER** all refer to the same named variable. Statement 5 appends the content of named variable **trailer** to the Album field. Statement 6 sets named variable **Preferred Genre** to **Blues**.

There are ten system variables numbered 0 to 9. System variables do not get removed when you close Yate. Whatever you put in them sticks around until you get rid of them. They are really only useful if you wish to keep some persistant information hanging around. You can examine the values of the system variables outside of actions in the application preferences System Variables panel.

———— Chapter 3 Snippet 3

```
4: Set System Variable 1 to "\<text to retain>"
5: Evaluate Expression "\v1+\s2" save result to Variable 4
```

————

Statement 4 sets system variable 1 to the contents of named variable **test to retain**. Statement 5 adds the values in track variable 1 and system variable 2 and saves the result to track variable 4. Note that both the variables have to contain a number.

As Yate has evolved and more complicated actions got written, it became apparent that a better method of retaining settings for actions was necessary. This lead us to the development of action runtime settings.

Action runtime settings are similar to named variables and system variables. However, they are not intended to be used as general purpose variables but rather a means of keeping state for an action after it terminates. Action runtime settings persist until they are removed. They cannot contain multi line data and in fact are truncated at the first, if any, newline character. While many statements can directly reference track and named variables, the same is not true for action runtime settings. The writing of an action runtime setting is somewhat less efficient than storing a named variable. While action runtime settings can be treated as named variables, it is not recommended that you do so. While they are persistent as are system variables, action runtime settings are symbolic and provide a well defined interface specifically designed for action settings. Action runtime settings tend to make system variables redundant.

Action runtime settings are stored in Runtime Settings sets. These sets can be viewed, modified and exported in Preferences - Runtime Settings. When an action starts, all loading and setting of action runtime settings is associated with a runtime setting set with the same name as the starting action. You can map to a different set by using the **Map Action Runtime Settings** statement. It is a good idea to store an action's settings in its own runtime setting set. Some actions may use multiple runtime setting sets. Action runtime settings ignore # character's in their name. This is because # has special meaning which will be presented in the next example.

By default when your set an action runtime setting to empty, the setting is removed and will not be visible in Preferences - Runtime Settings. In fact when all items in a runtime setting set are removed, the set is removed. This does not affect anything other than the reduction of the storage requirements. As is the case with named variables, a reference to an action runtime setting which does not exist will return empty. The following snippet describes some of the special means of modifying runtime settings.

———— Chapter 3 Snippet 4

```
4: Map Action Runtime Settings to 'Test Action's Settings'
5: Set action runtime setting 'initialized' to "1"
6: Set action runtime setting '#must retain' to ""
7: Remove all action runtime settings in the mapped set
8: Set named variable 'action is initialized' to "\#initialized#"
```

————

Statement 4 makes a runtime settings set name **Test Action's Settings** the source and destination for all subsequence action runtime setting references. Statement 5 sets a runtime setting named **initialized** to 1. Statement 6 prefixes the name of an action runtime setting with a # character. This is a special flag which causes empty action runtime settings not to be removed. **must retain** will have an empty value and will be visible in Preferences - Runtime Settings in whichever set was mapped. Statement 7 specifies the # character with no name and an empty value. (view the snippet in an action editor). This is a special form which causes all action runtime settings in the currently mapped set to be removed. Statement 8 sets an **action is initialized** named variable to the value of an action runtime setting named **initialized**.

You can easily copy all runtime settings to named variables of the same name as follows:

———— Chapter 3 Snippet 5

```
4: Get Info: Named Runtime Settings Set 'My Action Settings' as named variables *** The set does not exist
```

————

The Get Info statement is coping all setting in the **My Action Settings** runtime settings set to named variables of the same name. The *set does not exist* message is simply a warning that **My Action Settings** does not exist. This can be ignored as the set can be filled at runtime.

The problem with extracting all settings at once is that not all possible settings may currently have non empty values and as such may not be in the set. You can get around this issue by clearing all possible values before the extraction.

———— Chapter 3 Snippet 5

```
1: Clear named variables in list 'setting 1\nsetting 2\nsetting 3'
2: Get Info: Named Runtime Settings Set 'My Action Settings' as named variables *** The set does not exist
```

————

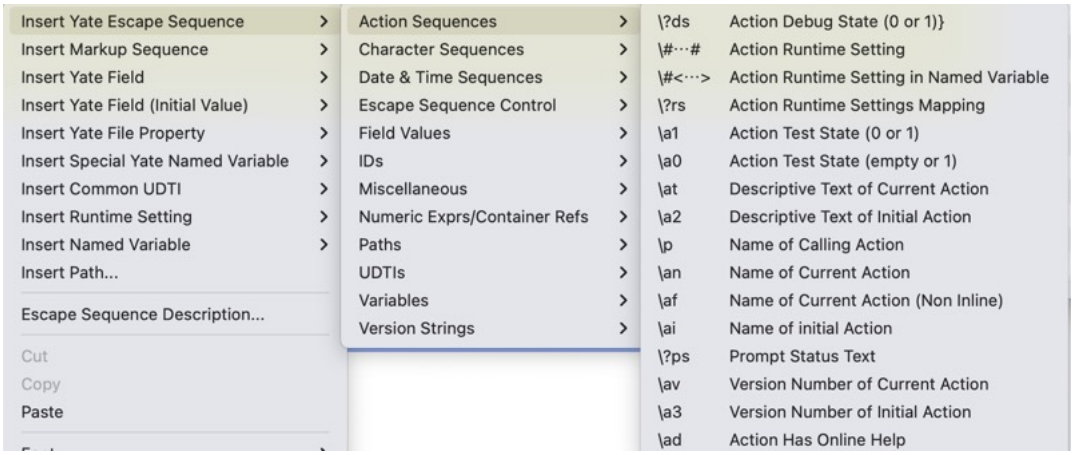
The Clear statement is configured to clear all named variables in a list. In this case named variables: setting 1, setting 2 and setting 3. If these are the names of the potential settings, clearing them will ensure that any settings read by the Get Info statement will have the correct value even if they are not present in the set.

Back to Table of Contents

Chapter 4 - Escape Sequences

Escape sequences are your friends! It is almost impossible to write effective actions without using escape sequences. Proper use of escape sequences allows you to perform an operation in a single statement as opposed to achieving the same effect with multiple statements. They are also the only way to access the *content* of named variables, system variables, action runtime settings and the value of the action test state. You can also access the initial value of fields before they were changed, properties and other useful content. Additionally you can evaluate numeric expressions in escape sequences.

Text fields in action statement definition panels have context menus to ease the insertion of escape sequences. *There are many of them*. There is also a context menu which essentially keeps track of the names of all named variables used in any open action ... and another which keeps track of referenced action runtime settings.



The menu items vary dependent of which action statement's field is being used. When you select a menu item, the escape sequence is inserted. You can even get a popup describing a text sequence already in a field. Simply place the cursor immediately before the \ character and select **Escape Sequence Description...** from the context menu.

Yate heavily relies on escape sequences to perform various functions directly within text fields in a statement's definition. The text fields in most Yate statements are unescaped at runtime to produce the final result. For example:

———— Chapter 4 Snippet 1

```
4: Set named variable 'timestamp' to "\z"
5: Test if the Album field is equal to "\{Album}" (Set test state)
6: Set the Comments field to "\!Bit Depth! - \!Sample Rate (Hz)! - \[BPM]"
```

————

Statement 4 sets named variable **timestamp** to the current time formated as follows: YYYY-MM-DDTHH:MM:SS.### (year-month-day T hour:minutes:seconds.milliseconds).

Statement 5 tests if the possibly modified value of the Album field is the same as its initial value (the value last written to the audio file).

Statement 6 statement sets the Comments field to a string composed of the bit depth and sample rate properties followed by the contents of the BPM field.

The Quick Help submenu on the Help menu has a link to the escape sequence definitions. You can also see it here.

Even we only know a fraction of the escape sequences without using the context menu. Use the context menu to access the sequences.

Back to Table of Contents

Chapter 5 - Execution Modes

Yate's execution modes are the toughest concept to grasp and possibly the most important. The two execution modes are *stepwise* and *grouped*.

A Yate action cannot execute without selected audio files. Actions can be written which do not access audio files but at least one file must be available. When you enable an action to run **always** in the Action Manager, a temporary file will be supplied if no files are loaded.

When executing *stepwise*, every statement is applied to each *active* audio file. Typically an active file is a selected file but there are action statements which can be used to change which files are active.

When executing *stepwise*, you can think of each file being processed in parallel. If you are executing *stepwise* and you issue a **Copy Album to Variable 1** statement, you will be copying each selected file's Album field to its unique Variable 1 field. One statement does the same thing with possibly different data, for each selected file. If a particular statement executed *stepwise* does something non standard, it will be described in its documentation. When action processing starts, the initial execution mode is *stepwise*. Some action statements, such as those that perform artwork retrieval, are more efficient when run *stepwise* as Yate can ensure that any one artwork item is downloaded or loaded from the filesystem, only once.

Many actions can be written and designed to be run entirely in *stepwise* execution mode. However, there are times when it is required to run *blocks* of code as a single entity for a single file before proceeding to the next file. When a section of code is run in its entirety for a single file, you are executing *grouped*. Typically *grouped* execution is required when conditional statements are used to control program flow. When making conditional decisions, different outcomes may be required for different files. The only *blocks* of code that can be run *grouped* are actions.

If you want to force an action to be executed *grouped*, you can do so by placing a **Force Grouped Execution** statement at the start of the action file. Use of a **Force Grouped Execution** statement is the only means of executing an action called directly from the UI *grouped*. Remember, action execution always starts *stepwise*. Be aware that from a speed point of view, this is not the most efficient way to run an action. Yate actions are interpreted and as such letting the interpreter handle multiple files at once in *stepwise* mode will execute far fewer statements.

Actions can be run from the UI or via the Batch Processor. When run via the Batch Processor, the action will be run on every folder which contains audio files. When starting the Batch Processor, one or more *root* folders are specified.

Back to Table of Contents

Chapter 6 - Action Structure

Actions can be separate entities, effectively files, contained in the Action Manager. These actions can be *called* from any other action. Actions can also be *inline*, contained inside an action *file*. Inline actions can only be called from within the action file which contains them. Unless otherwise directed, an action executes from its first statement to the end of an action file or until a **Start** statement is encountered. **Start** statements are used to specify the start of an inline action. There is no *end* statement for an action. You either hit the end of the file, or a Start statement.

I have a complicated action. Should I use inline actions or create multiple separate actions? Our general practice is that if an action can be reused, create it as a separate entity. If the action can be thought of as being used only within a specific action, create it inline. Many of the actions we supply are delivered as a single action file.

When an action is run through the Batch Processor, a search will be performed for two inline actions in the action *file* being run. The **Start Batch** inline action, if found, will be executed prior to the processing of any folders. The **Stop Batch** inline action, if found, will be executed after the last folder has been processed.

Back to Table of Contents

Chapter 7 - Flow Control: An Introduction

Actions execute linearly, one statement after another, unless another action is called or an **if** statement is executed. There is no limit to the number of actions which can be called, but recursion is not permitted. Any given action, file based or inline, can only be open for execution once, at any given time. There is also no limit to the number of nested **if-else-endif** constructs.

As initially implemented, an **if** statement could only test the **Action Test State**. The **Action Test State** contains a simple true or false value and is set by many action statements to report a successful or failed operation.

There are conditional test statements to compare *things* as text, integers or dates. These statements all set the **Action Test State**. These conditional test statements can **Set**, **And** or **Or** the test state. This is an efficient means of forming compound tests without requiring multiple **if-else-endif** constructs.

A statement which **Ands** the test state is only executed if the test state is currently true. A statement which **ORs** the test state only executes if the test state is currently false. A statement which **Sets** the test state always executes. Remember, statements are executed one after another. There is no precedence in Yate statement execution.

When using more than one statement to create a compound test, you may have to execute *grouped*. This is due to the fact that the test may produce different results for each file.

———— Chapter 7 Snippet 1

```
 4: Test if the Album Artist field is equal to "Various Artists" case insensitive (Set test state and Variable 0)
 5: Test if the Part of a Compilation field is true (Or test state and Variable 0)
 6: if true
 7:     ' statements for compilations
 8: else
 9:     ' statements for non compilations
10: endif
```

————

The above example tests if the Album Artist field is Various Artists (case insensitive) or if Part of a Compilation is true. The action test state is tested with an **if** statement which by use of an **else** statement executes one of two sequences of statements.

If not executing *grouped* the statement may be meaningless as different files might have different results and the test state may be indeterminate. As a matter of fact the default mode for the two **Compare Text** statements used above is that the test state is set to true if all files satisfy the test. If you want to handle the execution of different statements for each file you would have to execute the tests when grouped.

———— Chapter 7 Snippet 2

```
 4: Run inline action 'Compilation test' grouped
 5:
 6: Start Compilation test
 7: Test if the Album Artist field is equal to "Various Artists" case insensitive (Set test state and Variable 0)
 8: Test if the Part of a Compilation field is true (Or test state and Variable 0)
 9: if true
10:     ' statements for compilations
11: else
12:     ' statements for non compilations
13: endif
```

————

The above snippet uses a **Run** statement to call an inline action in *grouped* execution mode. This code will work correctly regardless of a file's settings for Album Artist and Part of a Compilation. However, this might result in hundreds if not thousands of unnecessary statements being executed depending on the number of active files. The snippet essentially turns off the parallel processing.

Various comparison statements including the **Compare Text** statement support the saving of the test result to a track variable as well as to the action test state. The Set, And, Or semantics are preserved when setting the track variable. This feature allows you to preserve individual test results on a per file basis while executing stepwise. After running any combination of these statements you can have a track variable set to true or false. The following example shows how this can work.

———— Chapter 7 Snippet 3

```
 4: Test if the Album Artist field is equal to "Various Artists" case insensitive (Set test state and Variable 0)
 5: Test if the Part of a Compilation field is true (Or test state and Variable 0)
 6: Run inline action 'Process compilations' if Variable 0 is true
 7: Logical set Variable 0 to NOT Variable 0
 8: Run inline action 'Process non compilations' if Variable 0 is true
 9:
10: Start Process compilations
11: ' statements for compilations
12:
13: Start Process non compilations
14: ' statements for non compilations
```

————

The above example tests the two conditions and saves the per file result to track variable 0. It then calls an inline action to process the compilations. The value in track variable 0 is then complemented and an inline action is called to process the non compilations. As these two **Run** statements do not specify *grouped*, they execute *stepwise*. This can possibly drastically reduce the number of statements which are executed.

Initially when an action starts all selected files are active and are processed when executing *stepwise*. Another way to get the desired effect would be to change which files are active during execution.

———— Chapter 7 Snippet 4

```
 4: Test if the Album Artist field is equal to "Various Artists" case insensitive (Set test state and Variable 0)
 5: Test if the Part of a Compilation field is true (Or test state and Variable 0)
 6: Ignore files where Variable 0 is false
 7: if true
 8:     ' statements for compilations
 9:     Restore Initial Set of Files
10: endif
11: Ignore files where Variable 0 is true
12: if true
13:     ' statements for non compilations
14:     Restore Initial Set of Files
15: endif
```

————

The above example sets the condition on a per file basis as we did before. However, the tests are immediately followed by an **Ignore Files** statement which ignores all files where Variable 0 is false (ie. not a compilation in the example). The **Ignore Files** statement sets the action test state to true if there is at least one remaining active file after its execution. Remember that Yate always requires at least one active file. You must always test the action test state after an **Ignore Files** statement to ensure that it worked otherwise the set of available files will not have been changed. You must also execute a **Restore Initial Set of Files** statement to restore the initial set of active files. The example uses this methodology twice to test for both conditions.

A Better Solution

The last few examples used various means to properly evaluate a condition and to execute as few statements as possible. While *file availability* statements such and **Ignore Files** and **Restore Initial Files** work they are cumbersome to use and require the manual adjustment and resetting of which files are available.

In Yate v6.16, the **if-else-endif** statement was enhanced to directly support the testing of track variables as well as the action test state. The improved **if-else-endif** statement solves the problem and automatically saves and restores which files are available automatically.

———— Chapter 7 Snippet 5

```
 4: Test if the Album Artist field is equal to "Various Artists" case insensitive (Set test state and Variable 0)
 5: Test if the Part of a Compilation field is true (Or test state and Variable 0)
 6: if Variable 0 is true
 7:     ' statements for compilations
 8: else
 9:     ' statements for non compilations
10: endif
```

————

The **if** statement on statement 3 tests the state of Variable 0 for every active file. All statements up until the **else** statement, (or **endif** if there is no **else**), are executed only on those files where Variable 0 is *true*. Statements from the **else** until the **endif** are executed only on those files where Variable 0 is *false*. Changing the test variable has no effect on the program flow once execution of the **if-else-endif** sequence starts. Which files are appropriate for each code section is determined when the **if** starts.

The need for *grouped* mode is eliminated as only *pertinent* files are available for each code segment. All manipulation of the set of active files is automatically handled but the net effect is the same. Calling out to another action from within the **if-else-endif** construct will have the files which were active when the action is called. You can use file availability statements such as **Restore Initial Files** within the construct but the appropriate files are reset automatically when the **else** and **endif** are executed. Further if you exit out of the construct, the appropriate files are always restored.

Back to Table of Contents

Chapter 8 - Flow Control: Running Actions

The **Run** statement is the simplest way to call another action. The statement can call inline actions and other action files. When an action is run it inherits the current execution mode unless otherwise directed. If **grouped** is specified, the designated action will always be run grouped. You'll note that there is no **stepwise** option. That's because you can never force stepwise execution. You can never run an action *stepwise* when you are already executing *grouped*. The **grouped** option does not change anything when you are already executing *grouped*. However, in our opinion, it is good practice to specify the **grouped** option when you want to run the specified action *grouped* even if already running *grouped*. It makes things a little clearer in the code.

There will be times when you want an action to operate on data completely outside of the selected audio files. You might be operating on data in a database file or on data stored in lists (later). In this case you do not want to run stepwise over every audio file and yet you do not want to run successively grouped on every audio file. The **once** option can be specified along with **grouped**. The effect is that the action will be run grouped but only once with a single audio file. If the choice of which audio file is active is pertinent, you shouldn't be running **once**. As already mentioned, there is no way to execute statements in Yate without having at least one audio file active.

A much rarer *run* option is **indirect**. When running indirect, the name of the action to be run is determined at runtime. This is useful when you want to run one of a number of possible actions dependent on certain conditions. The field defining the indirect action name should contain at least one escape sequence for a track or named variable which contains all or part of the action's name. When we get into lists it will become more evident how running *indirect* can enable a type of *switch-case* statement. You can extract the name of an action from a list based on various criteria and then execute the action indirectly.

The **Run** statement can also be told to run only for those files which have a track variable with a value of true. We've already used this configuration in one of the previous examples. This is useful when a compound test has saved individual test results to a track variable. The semantics are what you would expect for both *stepwise* and *grouped* execution modes. Regardless of the execution mode, only those files with the specified track variable having a value of true will get processed. Changing the *control* track variable in the called action will not change which files get processed. Note: The Run statement, when testing a track variable, is the only means of possibly executing an action zero times.

Here are some sample **Run** statements:

———— Chapter 8 Snippet 1

```
 4: Run action 'Yate and the 'P' Word'
 5: Run inline action 'Test'
 6: Run inline action 'Test' if Variable 0 is true
 7: Run inline action 'Test' grouped
 8: Run inline action 'Test' grouped if Variable 0 is true
 9: Run inline action '\<Action to Run>' indirect
10: Run inline action 'Test' grouped (once) if Variable 7 is true
11:
12: Start Test
13: ' ...
```

————

The sample **Run** statements cover what we have discused about the statement. The last statement presented is problematic. When only processing files which have a specific track variable being true, the **Once** option may not have the desired effect. **Once** is applied before the variable testing is performed. This means that if the first file tested does not have the variable with a value of true, the processing will be stopped. To handle this case do not use **Once** and place a Break statement in the action itself.

A **Break** statement stops execution of the current action, regardless as to how it was called or whether the execution mode is stepwise or grouped. The statement can be executed always or only if the action test state is true or false. When processed, the next statement to be executed is always the statement immediately following the caller.

———— Chapter 8 Snippet 2

```
4: Run inline action 'test' grouped if Variable 0 is true
5:
6: Start test
7: ' ...
8: Break
```

————

The above example runs inline action **test** on all active files where track variable 0 is true. The inline action **test** is terminated by a **Break** statement which causes execution to resume immediately after the **Run** statement.

Break is only one of a number of action statements which can be used to terminate action execution. All of these statements can be unconditional or can be executed only if the action test state is true or false. Further, the statements can optionally force a specific value to the action test state.

The most common of these statements is the **Exit** statement which is most analogous to the **return** statement found in most programming languages. **Exit** returns from a single instance of an action. When executing **grouped**, execution of the action will continue on the first statement of the action with the next selected file. When executing *stepwise*, or if there are no more files to process **grouped**, control will resume at the first statement following the calling statement. This statement mirrors what effectively happens if you encounter a **Start** statement or the end of an action while executing.

———— Chapter 8 Snippet 3

```
 4: Run inline action 'test'
 5:
 6: Start test
 7: Test if the Part of a Compilation field is true (Set test state and Variable 0)
 8: if Variable 0 is true
 9:     Set the Album Artist field to "Various Artists"
10:     Exit
11: endif
12: ' .....
```

————

In the above snippet action **test** tests if Part of a Compilation is true. For all files where it is true, the Album Artist is set to Various Artists and the action is terminated.

Let's assume that the above action should return an action test state of false if none of the files were Part of a Compilation. The above action could be modified as follows:

———— Chapter 8 Snippet 4

```
 4: Run inline action 'test'
 5:
 6: Start test
 7: Test if the Part of a Compilation field is true (Set test state and Variable 0)
 8: if Variable 0 is true
 9:     Set the Album Artist field to "Various Artists"
10:     Exit return false
11: endif
12: ' .....
13: Exit return true
```

————

Statements 10 and 13 in the example above, force the action test state to a particular value.

A description of the statements which can be used to terminate an action can be found in Chapter 9 - Flow Control: Exiting an Action

The **Test & Run** statement is a *run* variant which lets you test the **action test state**, run an action based on it being true or false, with all the options available on the **Run** statement and then continue, exit or stop action processing.

———— Chapter 8 Snippet 5

```
 4: Test if named variable 'initialized' is true (Set test state)
 5: If false, run inline action 'Perform Initialization' then continue
 6: ' Execute statements which require initialization
 7:
 8: Start Perform Initialization
 9: ' ...
10: Set named variable 'initialized' to "true"
```

————

The above snippet ensures that initialization has been performed by testing and executing the **Perform Initialization** action if required.

Actions can be repetitively executed. In its simplest form a **Repeat Forever** statement will continuously call an action until an **Exit Repeat** statement is executed. The **Repeat Forever** statement can only run inline actions.

———— Chapter 8 Snippet 6

```
 4: Repeat Forever wait for continue from user
 5: ' User caused action to continue
 6:
 7: Start wait for continue from user
 8: Test if the Option key is pressed (Set result)
 9: Exit Repeat if true
10: Pause '2' seconds
```

————

The above example waits for the user to hold down the Option key before continuing execution. The **Exit Repeat** statement cancels repeating when an action is called by a Repeat Forever statement.

A second repetitive execution statement is **Repeat With** which lets you execute an action based on components of a list. The statement calls the action once for each list element. The supplied list is copied so changing the list once execution starts will not change the execution flow. As an example you can choose to execute an action for each unique artist extracted from all selected files' Artist fields, treating items separated by commas in the individual Artist fields as separate items. There will be more on lists later. When executing *stepwise*, the action will be run once for every list element. When executing *grouped*, the action will be run once for each selected list element on each selected file. As with the **Run** statement, the **Repeat With** statement can be executed **grouped once**. In this case the action will be run once in grouped mode for each list item...but only once. As with the **Run** statement, this is useful when you want to run *grouped* but you're not working on data in an audio file or when the list components do not require further audio file interaction.

———— Chapter 8 Snippet 7

```
 4: Build List with delimiter "⏎" in named variable 'All Artists' from field Artist (As found) (case insensitive) string delimiter "\m"
 5: Build List with delimiter "⏎" in named variable 'All Album Artists' from field Album Artist (As found) (case insensitive) string delim
 6: List Manipulate take the case insensitive union of items in 'All Artists'(⏎) and 'All Album Artists'(⏎). Save to 'All Artists'(⏎)
 7: Sort the list in named variable 'All Artists' delimiter "⏎" as in Finder
 8: Repeat With named variable 'name' for the list in named variable 'All Artists' (As found) string delimiter "\~" Run inline action 'Pro
 9:
10: Start Process each artist
11: ' Named variable 'name' contains the name of a single artist
```

————

The first two statements use the **Build List** statement to build lists of all artist and album artists in the active files. A **List Manipulate** statement is then used to take the union of the two lists. The All Artists list is then used as the list to be processed by the **Repeat With** statement. The **Repeat With** statement will then call the **Process each artist** inline action for each unique artist name. The artist name will be in named variable **name**.

A third repetitive execution statement is **Repeat For** which is analogous to a typical programming language's **for** loop. The statement is given a start value, a stop value, a comparison operator and and an adjustment value. The execution options are the same as for the **Repeat With** statement.

———— Chapter 8 Snippet 8

```
4: Repeat For named variable 'index' = '0' while < '10' by '1' Run inline action 'Do something'
5:
6: Start Do something
7: ' Named variable 'index' contains 0, 1, 2, ... 9
```

————

The above **Repeat For** statement will call inline action **Do something** with named variable **index** containing 0, 1, 2, ...9

The last means of running an action is a special case statement where you want to load an audio file outside of the set of active files. The **Load and Run** statement, loads the specified audio file and executes an action on that file. The action can be inline and can be specified indirect. However the execution mode is always inherited. You cannot specify **grouped** and **once** is always implied. ie. you are always processing a single file. When the action finishes, the audio file will be closed. Note that the action more than likely should include a **Save** statement as changes are not automatically committed when the action is terminated. Any changes to the action test state in the called action are discarded. The statement sets the action test state to true if the file was successfully loaded, otherwise false.

———— Chapter 8 Snippet 9

```
 4: Load "\<File to Load>" and run inline action 'Process loaded file'
 5: if false
 6:     Prompt (Beep), "Failed to load audio file: \<File to Load>"
 7:     Exit
 8: endif
 9:
10: Start Process loaded file
11: ' The file loaded and is the only active file
```

————

An attempt will be made to load the audio file at the path in named variable **File to load**.

Back to Table of Contents

Chapter 9 - Flow Control: Exiting an Action

As already mentioned, actions execute until the end of the action file or until a **Start** statement is *hit*. There are a variety of action statements which can be executed to exit an action. The flow control effect of the exit depends on the statement. The following *return* statements are supported:

Break*
Break stops the execution of the current action, regardless as to how it was called or whether the execution mode is *stepwise* or *grouped*. The next statement to be executed is always the statement immediately following the caller.

Cancel
Regardless of the execution mode or where you are in a batch processing sequence, this statement stops all action execution. It is equivalent to hitting the Cancel button in the UI. Warning: modifications to open databases will not be automatically be saved when Cancel is executed.

Exit*
This statement returns from a single instance of an action. When executing **grouped**, execution of the action will continue on the first statement of the action with the next selected file. When executing *stepwise*, or if there are no more files to process **grouped**, control will resume at the first statement following the calling statement. This statement most clearly mirrors a typical return statement in higher level programming languages. This statement mirrors what effectively happens if you execute a **Start** statement.

Exit Grouped*
This statement stops the execution of all actions currently executing **grouped**. This means that you might be returning from more than one action. When executing *stepwise*, the statement is effectively identical to **Exit**.

Exit Repeat*
Regardless of the execution mode, if the currently executing action is a **Repeat Forever**, **Repeat For**, or **Repeat With** statement, all repeating will be stopped for all files. When not *repeating*, the statement is effectively identical to **Exit**.

Next File*
When executing **grouped**, the action immediately moves on to the next file to be processed. This means that you might be returning from more than one action. When executing *stepwise*, the statement is effectively identical to **Exit**.

Quit
This statement stops action processing and attempts to quit the application. Warning: All loaded files, hidden and visible, are closed without saving modifications.

Restart Batch Processor
This statement is only valid while Batch Processing and only in the **Batch End** inline action. The processing of folders will be restarted.

Restart Repeat Forever*
This statement conditionally stops the execution of the current action (regardless of execution mode), and flushes the call stack until it finds the last started Repeat Forever statement. If found, the Repeat Forever's action's first statement will be the next to execute. If not found, this statement is equivalent to **Stop**.

Stop
With one exception this statement stops all action processing. The exception is if you are batch processing. If you are executing the Batch Start inline action, control will proceed to folder processing. If you are processing folders, control will proceed to the next folder. If you're processing the last folder, control will proceed to the Batch End inline action.

Stop Action and Close Selected Files
This function will terminate action processing. It will then close all selected files as if the File>Close all Selected Files menu item was selected.

Stop Action and Filter Files
This statement immediately terminates action processing and will filter the list of open files according to the specified function, mode and filter.

Stop Action and Post Process
This statement immediately terminates action processing and can open and close selected files. It can even run a subsequent action.

Stop Action and Run Batch Processor
This statement immediately terminates action starts the Batch Processor based on supplied information.

Stop Batch Processing
This statement is effectively a **Stop** statement when not batch processing. When batch processing, it differs from **Stop** in that if processing folders, control will proceed immediately to the **Batch End** inline action.

With the exception of the **Stop Action...** statements, all of the above statements can be executed always or only if the action test state is true or false.

———— Chapter 9 Snippet 1

```
 4: Exit if true
 5:
 6: ' is the same as:
 7:
 8: if true
 9:     Exit
10: endif
```

———

Statements marked with an * can also specify that the action test state be set to true or false or toggled, if the statement is executed.

———— Chapter 9 Snippet 2

```
 4: Exit if true return false
 5:
 6: ' is the same as:
 7:
 8: if true
 9:     Set the action test state to False
10:     Exit
11: endif
```

———

Back to Table of Contents

Chapter 10 - File Availability

Please Read

> File Availability statements were implemented prior to the implementation of the **if Variable #** statement. They are a manual solution to the file availability problem. The **if Variable #** statement provides a virtually automatic solution for the same issues. As a lot of existing actions use these file availability statements, it is still worthwhile to understand how they work.

We've discussed the execution modes, conditional execution, running actions and briefly the **Ignore Files** and **Restore Files** statements. There are a few other means of changing how statements are executed.

Regardless as to what you are doing in an action, there must always be one active audio file. There is no exception to this rule.

When an action has its associated **Always** column checked in the Action Manager, it will always run. However, if no files are loaded or selected, one will be provided.

While not really a file availability statement, the **Force Grouped Execution** statement can be used to specify that an entire action is to be run grouped. If you don't want to worry about *stepwise* and *grouped* execution you can use this statement to force a single file to be processed at a time. However, this is almost always the least efficient method of executing an action. It also removes the capability of doing anything *stepwise*. Certain action statements such as the **Renumber Tracks** statement only support *stepwise* execution.

There are many actions which do not operate on a list of selected files. They might be interacting with a database or dynamically loading audio files when necessary. In these cases it is useful to specify a **Constrain Execution to a Single File** statement. Essentially it removes all but one of the current active files from consideration. This means that you can ensure that you are running *stepwise* but only have a single file. There are some special considerations as to when this statement is ignored:

- The action is executing *grouped*.
- A **Load & Run** statement is being executed.
- A *Run* statement is being executed which sets the file availability by testing a track variable.

The above conditions apply to all statements which modify the file availability

The effects of a **Constrain Execution to a Single File** statement can be undone by executing a **Restore Initial Set of Files** statement.

The **Ignore Files** statement is also ignored any time the tested condition would result in all files being ignored. For this reason it is important to always test if the statement succeeded so that you are not inadvertently processing files that you wanted to be inactive.

As an example assume you have many files loaded but that you only want to process files which have a genre of blues and are part of a compilation:

———— Chapter 10 Snippet 1

```
 4: Test if the Genre field is equal to "blues" case insensitive (Set test state and Variable 0)
 5: Test if the Part of a Compilation field is true (And test state and Variable 0)
 6: Ignore files where Variable 0 is false
 7: if true
 8:     ' Process the appropriate files
 9:     Restore Initial Set of Files
10: endif
```

————

The criteria was set up such that track variable 0 is true if both conditions are met. The **Ignore Files** statement ignores currently active files which do not meet the condition. By specifying an **if true** statement after the **Ignore Files** statement we can be sure that there are files to process. Finally we use a **Restore Initial Set of Files** statement to ensure that every file which was initially active is active again. The above example is a simplistic case. A much better implementation would be as follows:

———— Chapter 10 Snippet 2

```
1: Version 5.3@YatePWord
2: Cancel with failed run context
3:
4: Test if the Genre field is equal to "blues" case insensitive (Set test state and Variable 0)
5: Test if the Part of a Compilation field is true (And test state and Variable 0)
6: if Variable 0 is true
7:     ' Process the appropriate files
8: endif
```

————

You can successively ignore files if you wish. **Ignore Files** does not automatically restore the initial set before applying its criteria. **Restore Initial Set of Files** always restores the initial set, regardless as to how the set of active files was reduced.

You can take snapshots of the current active files and restore the same files at any time. The **Active Files to List** statement saves a list representing the files currently active. Do not attempt to manipulate the created list as more than likely it will not have the desired effect.

The **List to Active Files** statement restores the set of active files based on a list created by the **Active Files to List** statement.

These statements are useful when an action needs to modify the set of active files but cannot be sure that it is desired to restore the full initial set. As an example we'll modify the Chapter 10 Snippet 1 action.

———— Chapter 10 Snippet 3

```
 4: Save the list of active files to named variable 'save'
 5: Test if the Genre field is equal to "blues" case insensitive (Set test state and Variable 0)
 6: Test if the Part of a Compilation field is true (And test state and Variable 0)
 7: Ignore files where Variable 0 is false
 8: if true
 9:     ' Process the appropriate files
10:    Set the list of active files to the list in named variable 'save'
11: endif
```

————

There are times when you might want to make all loaded visible files active even if they were not initially selected. To do this you would use an **Expand Execution to Unselected Files** statement. To undo the effect of this statement you could use a **Restore Initial Set of Files** or **List to Active Files** statement.

Warning Please Read

If you change the file availability while executing under the control of a **Repeat With** or **Repeat For** statement, make sure you restore the available files to those available when the Repeat statement was started. If you do not, the execution flow may be unpredictable.

Back to Table of Contents

Chapter 11 - More on Lists

List processing is one of the more powerful features in Yate. The nature of tagging is such that you are quite often working on lists. For example, lists of artists, or composers, etc. The proper construction and use of lists can potentially drastically lower the number of action statements which must be executed. Yate code is interpreted and as such the fewer the statements the faster the code.

A list in Yate is simply an interpretation of a field or variable. Strictly speaking it is not a data type in and of itself. A list is interpreted based on its implied or supplied delimiter. Lists can be enumerated by the **Repeat With** statement. You can search, add to, remove from, build, sort and count the number of items in a list. You can get, set and remove individual list elements by index. Further, you can manipulate lists by their union, intersection, exclusion and by filtering. You can have lists of lists. Lists are the closest Yate concept to a high level programming language's array. You can also convert a list to and from CSV formats.

While lists are typically analogous to arrays, Yate supports the concept of key-value lists which are effectively a high level programming language's dictionary (or Object in JSON terms). In a key value list each list element has a key and an associated value. You can have values which are other lists or other key-value lists. All of this is possible by the selective choice of delimiters for each contained list.

Remember that lists are always simply strings in their implementation. You can mix and match list and text statements on the same data.

The **Build List** statement is used to construct lists. The source can be the contents of fields found in audio files or a named variable. When building lists based on the contents of fields, you can elect to only process those audio files which have a track variable with a value of true. This is analogous to the equivalent feature in various *Run* statements. The statement is extremely powerful and it's worth reading its online documentation for more information.

As an example let's say that you want to produce a list of all artists found in the Artist, Album Artist and Musician Credits fields.

———— Chapter 11 Snippet 1

```
4: Build List with delimiter "⏎" in named variable 'Artist List' from field Album Artist (As found) (case and diacritic insensitive) strin
5: Build List with delimiter "⏎" in named variable 'temp' from field Artist (As found) (case and diacritic insensitive) string delimiter "
6: List Manipulate take the case and diacritic insensitive union of items in 'Artist List'(⏎) and 'temp'(⏎). Save to 'Artist List'(⏎)
7: Save all people in Musician Credits to Variable 0
8: Build List with delimiter "⏎" in named variable 'temp' from field Variable 0 (As found) (case and diacritic insensitive) string delimit
9: List Manipulate take the case and diacritic insensitive union of items in 'Artist List'(⏎) and 'temp'(⏎). Save to 'Artist List'(⏎)
```

————

Statement 4 builds a list in named variable **Artist List** from every Album Artist field. The source delimiter is **\m** which is the multi-value delimiter (;;;). Statement 5 builds a list in named variable **temp** from every Artist field which delimits artists by the multi-value delimiter. Statement 6 takes the union of the two lists so that Artist List contains every unique artist from both the produced lists. Statement 7 is an **Involved People/Musician Credits Functions** statement which saves the names of each person in Musicians Credits to every files track variable 0. Statement 8 builds yet another list of every unique name in variable 0 in each file. This time the source delimiter is **\~** which is the default list delimiter (⏎) and was used when creating the lists in variable 0. Statement 9 takes the union of **Artist List** and the last produced list such that it now contains every unique artist name.

The following snippet uses a variety of list statements to display every genre referenced in all selected files. The action assumes that a comma is used to separated names in the Genre field.

———— Chapter 11 Snippet 2

```
4: Build List with delimiter "\n" in named variable 'Genre Counts' from field Genre (Counted) (case and diacritic insensitive) string deli
5: Find all matches for regular expression "≔(\d+)" in named variable 'Genre Counts', replace with " ($1)" to named variable 'Genre Counts
6: Sort the list in named variable 'Genre Counts' delimiter "\n" as in Finder
7: Prompt "<m><s>Genre Counts:</s>↵…"
```

————

Statement 4 builds a key-value list where the keys are the names of genres and the values are the counts. The list specifies that the delimiter is a newline character as we are going to display the results. This statement always uses **\k** (≔) as the key-value delimiter. Statement 5 uses a **Regular Expression** statement to change sequences **of ≔#** to **(#)**. Statement 6 sorts the list and Statement 7 displays the results.

Due to the nature of the **Build** statement, the previous snippet will not let us know if any actions had no genre. Build by default does not add empty items to the list. There is an option called Raw which can accumulate empty values but then the Genre field will be treated as a single item. We can get around the issue as follows:

———— Chapter 11 Snippet 3

```
 4: Build List with delimiter "\n" in named variable 'Genre Counts' from field Genre (Counted) (case and diacritic insensitive) string del
 5: Find all matches for regular expression "≔(\d+)" in named variable 'Genre Counts', replace with " ($1)" to named variable 'Genre Count
 6: Sort the list in named variable 'Genre Counts' delimiter "\n" as in Finder
 7: Test if the Genre field is empty trim newline characters (Set test state and Variable 0)
 8: Calculate the sum of the values in Variable 0 rounding (truncate). Result to named variable 'empty'
 9: Test if the integer value of named variable 'empty' == 0 (Set test state)
10: if false
11:     Add "<i>empty</i> (\<empty>)" to the end of the list in named variable 'Genre Counts' delimiter "\n"
12: endif
13: Prompt "<m><s>Genre Counts:</s>↵…"
```

———

Statement 7 sets track variable 0 to 1 if a file's Genre field is visually empty. Statement 8 sums the values in each file's variable 0. If the result is not zero, we add a line describing the number of files which have an empty Genre field.

If you find regular expressions daunting you can use simpler text statements to *prettify* the results.

——— Chapter 11 Snippet 4

```
 4: Build List with delimiter "\n" in named variable 'Genre Counts' from field Genre (Counted) (case and diacritic insensitive) string del
 5: Test if named variable 'Genre Counts' is empty (Set test state)
 6: if true
 7:     Replace "\k" in named variable 'Genre Counts' with "·("
 8:     Append ")" to named variable 'Genre Counts', as list with delimiter "\n"
 9:     Sort the list in named variable 'Genre Counts' delimiter "\n" as in Finder
10: endif
11: Test if the Genre field is empty trim newline characters (Set test state and Variable 0)
12: Calculate the sum of the values in Variable 0 rounding (truncate). Result to named variable 'empty'
13: Test if the integer value of named variable 'empty' == 0 (Set test state)
14: if false
15:     Add "<i>empty</i> (\<empty>)" to the end of the list in named variable 'Genre Counts' delimiter "\n"
16: endif
17: Prompt "<m><s>Genre Counts:</s>↵…"
```

———

The **Regular Expression** statement has been replaced with statements 5 thorough 10. Statement 5 tests if the list is empty. ie. there were no genres. If the list was not empty, statement 7 replaces the \k (≔) key-value separator with a space followed by a left parenthesis. The **Replace** statement can process lists but in this case it is simpler to simply change all matches in the string. Statement 8 uses an **Append** statement to append a right parenthesis to to every element in **Genre Counts**, treating it as a list.

You can use lists to implement a high level language's *switch* statement. You can do this with a numeric zero based index mechanism or by using a symbolic reference. The following snippet uses an index based mechanism to call one of three actions. A *default* is provided for out of bounds indexes.

——— Chapter 11 Snippet 5

```
 4: Set named variable 'Action List' to "Test 0,Test 1,Test 2"
 5: Repeat Forever Loop
 6:
 7: Start Loop
 8: Prompt for Text save to named variable 'index', "<m>Enter 0, 1 or 2 to run Test 0, Test 1 or Test2.↵…"
 9: Set named variable 'action' to the item at index '\<index>' of the list in named variable 'Action List' delimiter ","
10: if true
11:     Run inline action '\<action>' indirect
12: else
13:     Run inline action 'Test-Default'
14: endif
15:
16: Start Test 0
17: Prompt "In Test 0"
18: Start Test 1
19: Prompt "In Test 1"
20: Start Test 2
21: Prompt "In Test 2"
22: Start Test-Default
23: Prompt "In Test-Default"
```

———

The **List Item At Index/Sublist** statement on line 9 accepts negative integers to index from the end of the list. For that reason, the Prompt for Text statement on line 6 disallows negative numbers. (Click on the gear icon in the Prompt for Text statement's settings).

Here's a slightly modified version of the last example which uses symbolic references to implement the same thing.

——— Chapter 11 Snippet 6

```
 4: Set named variable 'Prompt for Text List' to "Test 0\~Test 1\~Test 2"
 5: Repeat Forever Loop
 6:
 7: Start Loop
 8: Prompt for Text save to named variable 'action', (Show picker), "<m>Select an action to run from the disclosure button's menu or enter
 9: Test if the list in named variable 'Prompt for Text List' delimiter "⏎" has an item equalling case insensitive "\<action>"
10: if true
11:     Run inline action '\<action>' indirect
12: else
13:     Run inline action 'Test-Default'
14: endif
15:
16: Start Test 0
17: Prompt "In Test 0"
18: Start Test 1
19: Prompt "In Test 1"
20: Start Test 2
21: Prompt "In Test 2"
22: Start Test-Default
23: Prompt "In Test-Default"
```

————

There are many more *list* statements and far more that you can with them. It is strongly recommended that you read the online help for the statements in the Lists action group in the action editor.

Back to Table of Contents

Chapter 12 - More on Containers

Containers being the only non text based data type in Yate represents quite a change. They were implemented to make it easier to interact with JSON data returned from **Open URL** statements. However, they allow you to represent any data model that you would like and to easily interact with it.

You can create, add items, set items, test for existence and evaluate datatypes easily by using the concept of a directed path. Container names are treated as case insensitive and cannot contain **.** or **[** characters.

A directed path is a string representation of how to traverse a container to identify a particular item. Successive array elements or object names are appended to the list. For example:

test[1].address.streetNumber

The above references an object member named **streetNumber** in another object member named **address** in an object at index 1 in the container **test** which is an array. The following json content could represent sample data stored in **test**:

```
[
  {
    "address" : {
      "city" : "Any Town",
      "street" : "Main Street",
      "streetNumber" : "123"
    },
    "name" : "John Doe"
  },
  {
    "address" : {
      "city" : "Any Town",
      "street" : "West Avenue",
      "streetNumber" : "456"
    },
    "name" : "Jane Doe"
  },
  {
    "address" : {
      "city" : "Any Town",
      "street" : "East Avenue",
      "streetNumber" : "789"
    },
    "name" : "Jim Doe"
  }
]
```

test[1].address.streetNumber would reference **456**.

While container names are case insensitive object member names are not. This is to remain compatible with JSON. Note also that object member names can contain any character including spaces. If a member name contains a space, period (.) or left square bracket ([), you have to enclose the name in double quote (") characters. For example if we were using **street number** as opposed to **streetNumber** the directed path would be: **test[1].address."street number"**. If a name contains a double quote character you can represent it as two concurrent double quote characters: ""

Here's a snippet using the various statements which can create a container:

———— Chapter 12 Snippet 1

```
4: Create container 'test' from JSON text '[ .... ]'
5: Create empty array container 'test2'
6: Create empty object container 'test3'
7: Create container 'test4' as a copy of the container item at 'test[2].address'
8: Create container 'test5' from JSON text in named variable 'raw json data'
```

————

Statement 4 is a **Create Container from JSON** statement and creates a container named **test** with the sample JSON text displayed above. Statement 5 is a **Create Array Container** statement which is creating an empty container named **test2** which is an array. Statement 6 is a **Create Object Container** statement which is creating an empty container named **test3** which is an object. Statement 7 is a **Create Container from Container Item** statement. This statement creates a new container which is a copy of all or part of a different container. The directed path in the statement must resolve to be an array or object. In this case test 4 would represent the following data:

```
{
  "city" : "Any Town",
  "street" : "East Avenue",
  "streetNumber" : "789"
}
```

Statement 8 is another **Create Container from JSON** statement which takes its input from a named variable. You would use this variant if you had downloaded JSON data via an **Open URL** statement.

Yate provides the Container Viewer which lets you expand and collapse individual elements in a container.

The **Debug** statement can display the contents of a container in the Log Viewer or in the Container Viewer. The **Display Container Item in Log Viewer** statement will display all or part of a container in the Log Viewer as JSON text. The directed path must reference an array or object. The **Display Container Item in Container Viewer** will display all or part of a container in the Container Viewer. Again, the directed path must reference an array or object. Here's a snippet that you can run to show both statements in action:

———— Chapter 12 Snippet 2

```
4: Create container 'test' from JSON text '[ .... ]'
5: Display the container item at 'test' in the Log Viewer
6: Display the container item at 'test[1]' in the Container Viewer
```

————

Statement 5 displays the entire **test** container in the Log Viewer. Statement 6 displays the object at **test[1]** in the Container Viewer.

If for any reason you want access to the JSON representation of a container you can use the **Extract JSON Text from Container** statement. The statement's directed path must reference an array or object. The referenced item will be saved as JSON to a specified named variable. The statement has the ability to produce formatted (default) or compressed text. The following example shows a a sample usage and displays the produced compressed text in the log viewer.

———— Chapter 12 Snippet 3

```
4: Create container 'test' from JSON text '[ .... ]'
5: Extract JSON text from the container item at 'test[1]' to named variable 'JSON Text', compressed
6: Show the contents of named variable 'JSON Text' in the Log Viewer (monospace)
```

————

When referencing items at a directed path, it is considered an error if the item does not exist. All *container* statements have a setting to optionally terminate an action if an error occurs. The problem of an item not existing can easily be resolved by simply not setting the option to terminate on an error. If you are extracting a value and it does not exist the result will be empty. Further, all *container* statements set the action test state to true on success and false on failure (when not terminating). That being said, you can use the **Test if Container Item Exists** statement to test if a container or any component exists. The following example performs some tests and displays the data in the Log Viewer and the test results in a prompt. A value of 1 means *exists* while a value of 0 means *does not exist*.

———— Chapter 12 Snippet 4

```
 4: Create container 'test' from JSON text '[ .... ]'
 5: Test if the container item at 'test' exists, stop on error
 6: Set named variable 'test results' to "\a1"
 7: Test if the container item at 'test[0].address.street' exists, stop on error
 8: Set named variable 'test[0].address.street results' to "\a1"
 9: Test if the container item at 'test[4]' exists, stop on error
10: Set named variable 'test[4] results' to "\a1"
11: Test if the container item at 'test[0].data' exists, stop on error
12: Set named variable 'test[0].data results' to "\a1"
13: Display the container item at 'test' in the Log Viewer
14: Prompt "test results: \<test results>↵…"
```

————

Items can be added to the end of an array by the **Add Item to Container Array** statement and object and array items can be set (or overwritten) by the **Set Item in Container** statement. When setting an array item, there is a special form of the directed path which allows you to insert an item as opposed to overwriting it. Precede the insertion point index with an **@** character. eg. obj.list[@5] will insert the specified content at index 5 as opposed to overwriting the item at index 5.

Items can be removed by the **Remove Item in Container** statement. The following example removes adds and sets items. The original and modified containers are displayed in the Log Viewer and in the Container Viewer. Run the action to see the results.

———— Chapter 12 Snippet 5

```
 4: Create container 'test' from JSON text '[ .... ]'
 5: Create container 'original' as a copy of the container item at 'test'
 6: Extract JSON text from the container item at 'test' to named variable 'data'
 7: Prepend "Original data in container 'test:\n\n" to named variable 'data'
 8: Append "\n\nModified data:\n\n" to named variable 'data'
 9: Remove the container item at 'test[1]'
10: Add JSON text '{ .... }' to the container array at 'test'
11: Set the container object item at 'test[0].moved' to boolean 'true'
12: Extract JSON text from the container item at 'test' to named variable 'modified'
13: Append "\<modified>" to named variable 'data'
14: Show the contents of named variable 'data' in the Log Viewer (monospace)
15: Create empty object container 'new'
16: Set the container object item at 'new.original' to referenced item 'original'
17: Set the container object item at 'new.modified' to referenced item 'test'
18: Display the container item at 'new' in the Container Viewer
```

————

With both of the **Add Item to Container Array** and **Set Item in Container** statements you must indicate the type of data that you are adding or setting. The following data types are common to both statements:

JSON Text
: You're adding or setting JSON text specified in the statement or in a named variable.

Referenced Item
: You're adding or setting a copy of a container or a portion of a container. The item is specified in a directed path.

Empty Array, Empty Object
: You're adding or setting an empty *compound* item.

Null, Number, String, Boolean
: You're adding or settinga *simple* data item.

The **Add Item to Container Array** has an additional data type that can be added:

Items in Array
: The array path must reference an array. All items in the referenced array are added to the end of the array specified in the directed path field. This statement can be interesting as all added items are copied. This means you can add an array to itself or all of an array to part of a contained array.

We've already discussed the **Remove Item in Container** statement which can be used to remove an entire container. It is a good idea to remove a container when you no longer need it as it will persist until the action ends ... or in the case of the Batch Processor, until batch processing is terminated. The **Remove All Containers** statement will remove all created containers.

The last statement to discuss is the **Extract Data from Container** statement. This statement is used to extract data at a supplied directed path. When extracting an array the number of elements in the array is returned. When extracting an object, the names of each named item in the object is returned as a list with items separated by the default list delimiter (\~). The following example extracts each possible type of data and displays the JSON text in the Log Viewer and the results in a prompt. Note that a **null** object returns an empty value.

———— Chapter 12 Snippet 6

```
 4: Create container 'test' from JSON text '[ .... ]'
 5: Display the container item at 'test' in the Log Viewer
 6: Extract data from the container item at 'test' to named variable 'test result'
 7: Extract data from the container item at 'test[1]' to named variable 'test[1] result'
 8: Extract data from the container item at 'test[1].address' to named variable 'test[1].address result'
 9: Extract data from the container item at 'test[1].address.street' to named variable 'test[1].address.street result'
10: Extract data from the container item at 'test[2].sampleBoolean' to named variable 'test[2].sampleBoolean result'
11: Extract data from the container item at 'test[2].sampleNull' to named variable 'test[2].sampleNull result'
12: Extract data from the container item at 'test[2].sampleNumber' to named variable 'test[2].sampleNumber result'
13: Prompt "test result: \<test result>↵…"
```

————

It is entirely possible that items can exist in a container and you do not know the data type of the data. For that reason the **Extract Data from Container** statement has an option to return a data type tag before each returned item. The data type tags for each type are as follows:

@ for String
# for Number
! for Boolean
~ for Null
{ for Object
[for Array

The next example is the same as snippet 6 with the exception that every statement is returning its data type. The data type characters are displayed bold-red in the prompt statement.

———— Chapter 12 Snippet 7

```
 4: Create container 'test' from JSON text '[ .... ]'
 5: Display the container item at 'test' in the Log Viewer
 6: Extract data from the container item at 'test' to named variable 'test result', include data tag
 7: Extract data from the container item at 'test[1]' to named variable 'test[1] result', include data tag
 8: Extract data from the container item at 'test[1].address' to named variable 'test[1].address result', include data tag
 9: Extract data from the container item at 'test[1].address.street' to named variable 'test[1].address.street result', include data tag
10: Extract data from the container item at 'test[2].sampleBoolean' to named variable 'test[2].sampleBoolean result', include data tag
11: Extract data from the container item at 'test[2].sampleNull' to named variable 'test[2].sampleNull result', include data tag
12: Extract data from the container item at 'test[2].sampleNumber' to named variable 'test[2].sampleNumber result', include data tag
13: Set named variable 'all nvars' to "\?nv"
14: List Manipulate Filter items in 'all nvars'(⏎) using text starts with "test". Save to 'all nvars'(⏎)
15: Repeat With named variable 'nvar' for the list in named variable 'all nvars' (As found) string delimiter "\~" Run inline action 'Highl
16: Prompt "<m>test result: \<test result>↵…"
17:
18: Start Highlight One Result
19: Set named variable 'text' to "\@<nvar>"
20: Substring [0, 1] of named variable 'text' -> named variable 'data type'
21: Substring from index '1' of named variable 'text' -> named variable 'result'
22: Set named variable '\<nvar>' to "<s><r>\<data type></s>\<result>"
```

————

The **Extract Data from Container** statement has the ability to verify the data type is as expected when extracting data. If the data type is not as expected, the action test state will be set to false. You can test for Any (which is the default), Array, Object, Number, Null, String, Boolean and Node. Node will return true if the referenced data is anything but an Array or Object. Number will match Number or Boolean data. If the required data type is Number and the actual data is Boolean, a value of 1 or 0 will be returned as opposed to true or false.

The **Extract Data from Container** statement supports a special form of a directed path to extract all components of an array. The form [*] implies that a referenced array is to be enumerated. At most one [*] form may be specified. The List Secondary Delimiter \: (●) separates the returned array elements. Note that if the array does not contain any elements, the destination named variable will be empty. If an item does not exist it will be returned as empty. If data tags are being returned a tag of **?** will be used to represent an item which does not exist. The following example returns every **street** item in the **test** container.

———— Chapter 12 Snippet 8

```
4: Create container 'test' from JSON text '[ .... ]'
5: Display the container item at 'test' in the Log Viewer
6: Extract data from the container item at 'test[*].address.street' to named variable 'result no data tag'
7: Extract data from the container item at 'test[*].address.street' to named variable 'result with data tag', include data tag
8: Prompt "result no data tag:↵…"
```

————

The only remaining container statement is the **Prompt from Container** statement which can be used to design a UI panel with a ton of configuration options. The statement and its container fields are documented in the online help.

Back to Table of Contents

Chapter 13 - Text Files and Databases

Yate can read and write text files. You can use statements to convert CSV databases to lists and convert lists to CSV data to be written.

Using Text statements such as **Scanner** you can parse virtually any text file. The **Repair Files from Audio File Health Check Log** action parses the plain text file produced by the **Audio File Health Check** application. Virtually any text file containing metadata can be read and applied to specified files.

Databases can be created, queried and manipulated. The best way to learn about databases is to see how they are used in some of the sample actions on the resources web page. One good example is the **Search Discogs/MusicBrainz** suite of actions.

Back to Table of Contents

Chapter 14 - Editing States

Yate maintains three versions of the metadata in a file. The **initial state** represents what was in the file when first loaded or after being saved. The **current state** represents the current, possibly unsaved, metadata. The **editing state** represents the metadata at a particular point in time (think snapshot).

In the UI the editing state can be manually and automatically updated. Also you can restore to the editing state as opposed to the initial values.

In actions you can use the **Update Editing State** and **Restore to Editing State** statements to maintain and restore to snapshots of the metadata.

An interesting point to remember is that while the **initial state** is reset when a file is saved, the **editing state** is not. It is therefore possible to revert to a state before files were saved.

As initial values are available for fields you can use escape sequences to access the initial values. You can generate the escape sequences via a text field's context menu **Insert Yate Initial Field** submenu.

Back to Table of Contents

Chapter 15 - Extracting Information

Yate retains many different kinds of information. The Get Info statement allows you to extract most of it. This information includes lists of fields (of various types); preference lists such as the Artist list; the contents of tag sets; the contents of rename and exception templates; lists of named variables and much more.

Earlier versions of Yate had two statements named Enumerate Fields, Roles and Extensions and Extract Preference Set. In fact, those statements were renamed many times over the years. Yate v6.13 merged the two statements in a single Get Info statement and added additional functionality.

Back to Table of Contents

Chapter 16 - Debugging

The only real method of debugging an action is to use the *old school* method of printing things. While you can use any statement in the Prompt family, the **Debug** statement was designed for this purpose. You can pick and or construct (with escape sequences) a message to be displayed.

At runtime you can display any combination of:

- The action test state
- the debug test state
- the call stack
- the execution mode
- the files initialled selected when the action was run
- the files currently active
- a snapshot of the track variables in active files
- per file track variables
- system variables
- named variables
- runtime settings
- a list of all open databases
- a list of all active containers

Debug statements can also perform the following functions:

- Open any action
- Open and position an action to any location on the call stack
- Display the initial or current metadata of files in the Log Viewer
- View the contents of a container in the Container Viewer or Log Viewer

Debug statements can display conditionally by testing the action test state or the debug test state. The debug test state is simply a retained state which can be conditionally set and tested by the Debug statement. The advantage of using the debug test state over the action test state is that you can set up Debug scenarios without modifying the action test state in the action. When you close a Debug panel, you can optionally set or clear the debug test state.

The Compare Dates, Compare Numeric and Compare Text statements can be configured to modify the debug test state as opposed to the action test state. This allows you to conditionally display Debug panels under virtually any circumstances. When one of these *compare* statements is configured to modify the debug test state, it is treated as a *debug* statement. This means it is displayed with red text and will be found by a Find - Debug search.

Action statements can be disabled and re-enabled from the context menu in an Action Editor window. This might make things a little easier while debugging as you can leave *debug* statements in the file until you are done.

Back to Table of Contents

Summary

We know that there are a few strange programming concepts here. However, they are really necessary to implement a diverse range of actions from the trivial to the extremely complicated.

Additional information, including sample actions can be found at Yate Resources. You can also visit the forum.

Don't stress, take your time and play around....while listening to some calming music.

Back to Table of Contents

Document History

Date	Version	Information
2016-09-07	1.0	First release.
2017-05-03	1.1	Added information on the **Repeat For** statement.
2017-10-17	1.2	Described the options available on all *return* statements. **Exit**, **Cancel**, etc.
2017-12-10	1.3	Described the ability to save test results to variables and to run grouped only on files if a specific variable is true. Fleshed out some other sections.
2018-04-03	2.0	Updated for changes in Yate v4
2018-04-25	2.1	Updated for changes in Yate v4.1
2018-05-22	2.2	Minor changes.
2018-09-13	2.3	Updated to discuss a few additional flow control statements.
2019-02-02	2.4	Minor changes.
2019-03-25	2.5	Minor changes.
2019-09-09	3.0	Updated for Yate v5.0
2020-06-03	4.0	Updated for Yate v6.0
2021-11-01	5.0	Major redesign. Released with Yate v6.7
2023-01-02	5.1	Updated for Yate v6.13
2023-01-29	5.2	Additional information on code snippets.
2023-06-30	5.3	Updated for **if Variable #** statements and setting the action test state on various *exit* statements introduced in Yate v6.16