Welcome to the Cleverscript Manual and Tutorial. This manual will show you how to create a very clever script in a spreadsheet. You can follow it in order, or dip in and out, or just leave the advanced examples till later. Within just a few minutes, you'll be able to have a chat with your very own bot. By the end, they will be very powerful and complex. The fifth section includes a basic informational bot and game bot. The sixth section explains how to integrate some small-talk into your Cleverscript, based on the intelligence behind Cleverbot. The last section introduces our website and API. Visit www.cleverscript.com to get started.

Contents

Section 1 – Inputs and outputs
Example 1.1 – Repetitive Bot
Example 1.2 – Hello and Sorry
Example 1.3 – An Interesting Discussion about Peas
Example 1.4 – One Line Input/Outputs

Section 2 – Phrases
Example 2.1 – Lots of Hellos
Example 2.2 – Introducing Phrases
Example 2.3 – Rapidly Expanding Phrases
Phrasing Guidelines
Example 2.4 – Output Modes and Capitalisation
Advanced Example 2.5 – Output phrases
Advanced Example 2.6 – Percentages within Phrases
Advanced Example 2.7 – Optional Phrases and Text
Advanced Example 2.8 – Unimportant Phrases
Advanced Example 2.9 – Wildcards

Section 3 – Learning
Example 3.1 – Basic Learning
Example 3.2 – Learning from Input
Example 3.3 – Using What You've Learned
Internal Variables
Example 3.4 – Learning Multiply and Mathematically
Example 3.5 – Learning Shortcut in Phrases
Advanced Example 3.6 – Calculating and Overloading
Advanced Example 3.7 – Learning with Wildcards
Advanced Example 3.8 – Dynamic Variables, Arrays and Temporary Variables
Advanced Example 3.9 – Parameters
Advanced Example 3.A – Learning from Phrases

Section 4 – If Conditions
Example 4.1 – Conditional Outputs
Example 4.2 – Debugging and Blanks
Advanced Example 4.3 – Complex Conditions
Advanced Example 4.4 – Variables in Ifs and Learns

Section 5 – Structure
Example 5.1 – Multiple Gotos
Example 5.2 – Dynamic and Unhandled Gotos – Informational Bot
Example 5.3 – Dynamic Phrase Labels
Example 5.4 – Conditional Inputs – Game Bot
Example 5.5 – Output Borrowing
Example 5.6 – Conditional Outputs
Example 5.7 – If and Learning Summary
Structural Approaches

Section 6 – Clever Data for Small Talk
Example 6.1 – Clever Data Fallback
Example 6.2 – Clever Data Variables
Example 6.3 – Reactions and Emotions
Advanced Example 6.4 – Influencing Clever Data
Advanced Example 6.5 – Influencing Clever Data Even More

Section 7 – Filtering
Example 7.1 – Input Filtering to Remove Text
Example 7.2 – Filters Between Inputs
Example 7.3 – Input Filtering to Replace Text
Example 7.4 – Output Filtering

Section 8 – Testing and Using Your Bot
8.1 – Register and Login
8.2 – Manual
8.3 – Upload Your Spreadsheet
8.4 – Chat to Your Bot
8.5 – Buy Credits
8.6 – Publishing
8.7 – Javascript API
8.8 – Usage Statistics
8.9 – Android and iOS APIs
8.10 – Self Hosting

Section 1 – Inputs & Outputs

This section will guide you through setting up your very first (second and third) bots. A "bot" is what we call the artificial entity which you will be creating and chatting with. It's short for "robot".

Example 1.1 – Repetitive Bot

Start Excel or Open Office or Numbers or Google Docs or whatever spreadsheet software you prefer, and type the following column headings:

Column A: Type
Column B: Label
Column C: Description
Column D: Text
Column E: If
Column F: Learn
Column G: Goto
Column H: Accuracy

The minimum which any bot needs is a single *output start* line. The *output start* is what the bot says to you at the very start of a conversation. Add a single row to your spreadsheet with some *Text* for the bot. Like this:

Type	Label	Description	Text	If	Learn	Goto	Accuracy
output start	welcome	The first thing the bot will say.	Hi there, I'm your very first repetitive bot!				

Your bot will only know how to output one thing: "Hi there, I'm your very first repetitive bot!" So that's what it will say at the beginning of your conversation. Then it will wait for your reply. And whatever you say, it will repeat its single line. The conversation will quickly get boring, but it's a start.

To try this for yourself, save your spreadsheet as a **tab delimited file**. It is very important that it uses tabs and not commas as a delimiter. Depending on your spreadsheet software, the file ending may be txt (when using Google Docs), tsv (some versions of Microsoft Office) or csv (OpenOffice). See section 8.3 of this manual for instructions on converting your spreadsheet into tab-delimited format.

If you have any non-Latin characters in your spreadsheet, please save with the character encoding **UTF-8**. A limited number of other encodings are accepted, but UTF-8 is the easiest to handle. Then login and register at www.cleverscript.com, upload your spreadsheet, check for any errors and start chatting to your new bot. This process is fully described in the last section of this manual.

Important points:

- all bots must have an *output start*, it's the first thing the bot will say
- save your spreadsheet as a **tab delimited file** with character encoding **UTF-8**
- register and login at www.cleverscript.com and upload your spreadsheet
- all Cleverscript outputs are limited to 255 characters maximum length

Example 1.2 – Hello and Sorry

This example will show you how to give a different *output* depending on the user *input*. The *input* is what you say back to the bot. Creating a bot involves predicting the different things the user might say and responding appropriately. Add the two *inputs* and two extra *outputs* below, and try to figure out what they do:

Type	Label	Description	Text	If	Learn	Goto	Accuracy
output start	welcome	The first thing the bot will say.	Hi there, I'm your second apologetic bot!				
input	hello	The user says hello	hello			hello_back	75
input	anything	Anything else	anything			sorry	0
output	hello_back	The bot says hello back to the user.	Hello to you too.				
output	sorry	Bot apologises for not knowing what to say.	Sorry, I don't have a response for that.				

After your bot outputs its starter “Hi there, I'm your second apologetic bot!” it will wait for your reply. It will then compare your *input* with its two *input* lines. If you say something 75% similar to the *Text* “hello”, then the bot will *Goto* the *output* labelled *hello_back*. Or else, if you say something 0% similar to the *Text* “anything”, it will go the *output* labelled *sorry*.

The *Accuracy* column determines how accurate your prediction needs to be. If you set the accuracy of the *input* to 100%, then only the exact word “hello” would match. If you set it to something really low like 20% then all sorts of things, like “heeeeelellllloooo” would also match. 75% allows for some typos and mistakes so “helloo” would also match. Note that upper and lower case and any punctuation marks at the end are ignored when making string comparisons.

The 0% on the second *input* means it will match anything else. So anything not 75% similar to “hello” will *Goto* the *output* labelled *sorry*.

If you removed the *input* labelled *anything* and then gave nonsense input to the bot, it wouldn't know what to do, so it would just go back to the beginning and repeat “Hi there, I'm your second apologetic bot!” That's essentially what was happening in the first example. The bot was saying its only line, waiting for the *input*, not making any sense of it, and so going back to the *output start* and repeating “Hi there, I'm your very first repetitive bot!”

Important points:

- *input* lines match user inputs
- upper and lower case and trailing punctuation marks are ignored when comparing text
- the *Accuracy* column determines how accurate the match must be
- if you leave the *Accuracy* blank, it defaults to 75%
- the *Goto* column tells the bot which *output* to go to after a successful match

Example 1.3 – An Interesting Discussion about Peas

This example starts to create a proper conversational script, where the *input* depends on the *output* and the *output* depends on the *input*. The lines labelled *anything* and *sorry* are not shown this time (for brevity) and two new *inputs* and *outputs* have been added:

Type	Label	Description	Text	If	Learn	Goto	Accuracy
output start	welcome	The first thing the bot will say.	Hi there, I'm your third pea-crazy bot!				
input	hello	The user says hello	hello			like_peas	75
output	like_peas	The bot asks a question	Do you like peas?				
input	peas_yes	User says yes	Yes			peas_yes	20
input	peas_no	User says no	No			peas_no	20
output	peas_yes	Bot agrees	Great! Me too!				
output	peas_no	Bot disagrees	That's a shame.				
output	blank_input	Reply to blanks.	I can't hear you.				

In this case, when the user (you) says “hello”, the bot goes to the *output* labelled *like_peas* and asks “Do you like peas?” This *output* line has two of its very own *inputs* for *peas_yes* and *peas_no*. If you reply “Yes” with only 20% accuracy, the bot will go to its *output* labelled *peas_yes*. And if you say “No”, it will go to *peas_no*.

The order in which the *inputs* and *outputs* appear in this spreadsheet is very important. The *input* labelled *hello* appears under an *output start*, therefore it is always active. No matter where you are in the conversation, you can say “hello”, and the *input* labelled *hello* will take over and direct you to the *output* labelled *like_peas*.

However, the *inputs* labelled *peas_yes* and *peas_no* do not appear under the *output start*. They appear under the normal *output* labelled *like_peas*. Therefore, they are only active after the bot says “do you like peas?”. If you say “Yes” after the bot says “Hi there I'm your third pea-crazy bot!” or “That's a shame”, it will not be matched, and the bot will repeat its opening line (or would have said “Sorry I don't have a response for that” if we had left that in).

This allows you to build a bot with a branching structure, as shown in the digram below. In summary *inputs* under an *output start* are always active, but *inputs* under a normal *output* are not. They belong to a specific *output* and are only active just after the bot has said that *output*.

Note that 20% is a low accuracy. It means that anything remotely resembling Yes, such as “Yeah dude” will match. And the same for “Not really”. Only if you say something really different like “You gotta be kidding.” will neither match, and the bot will return to its starting line. The danger here is that inputs like “none of your business” will produce false matches (26% match to “No”). Fortunately, Cleverscript has a great way to deal with this, starting from the next example.

This example also shows that an *input* and an *output* are allowed to have the same label. In fact, the *Goto* is unnecessary in this case. If there is no *Goto* the bot will look for an *output* with the same label as the matched *input*.

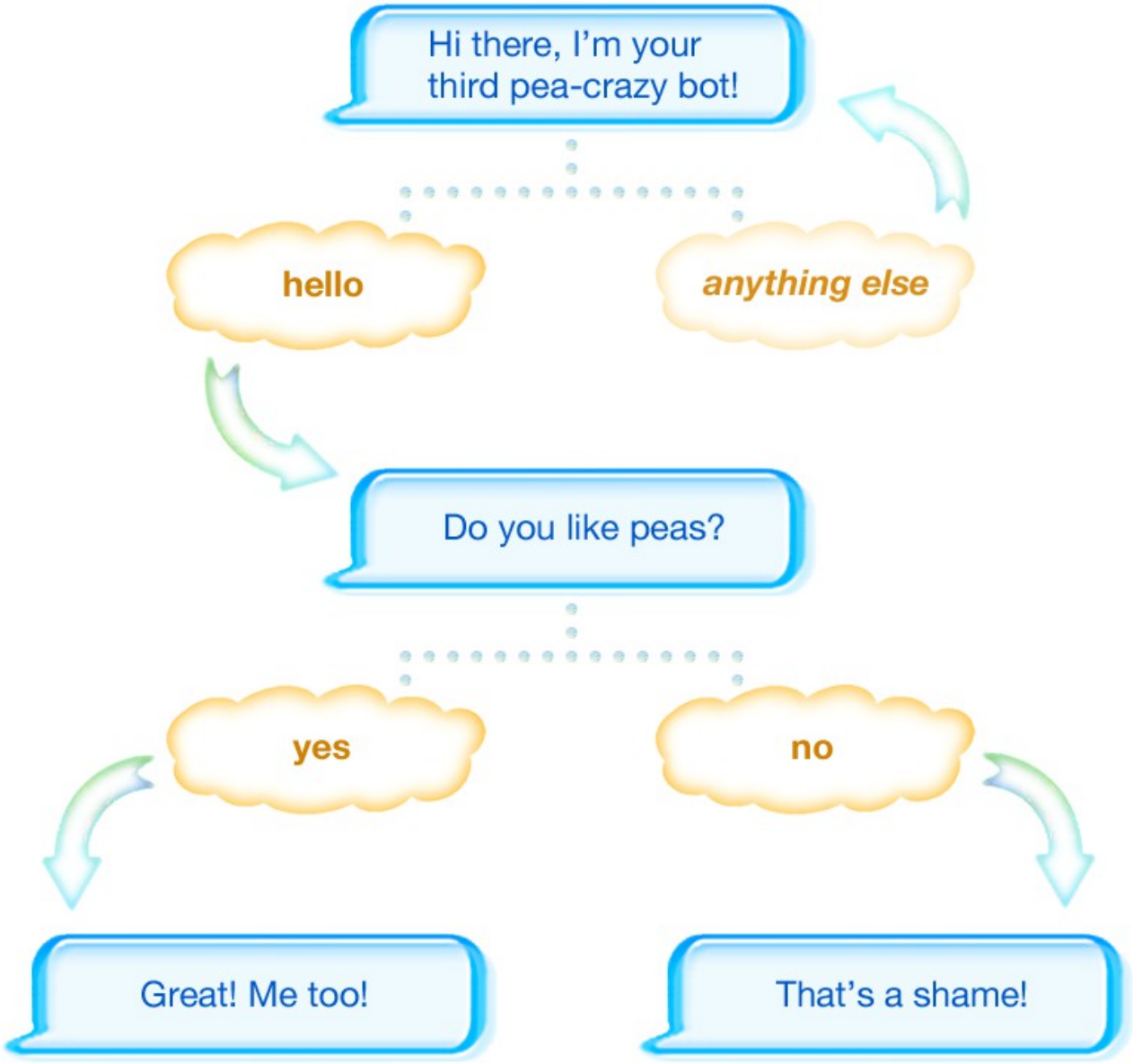
Otherwise, the labels within the bot must be unique. That's why I didn't just label the “Yes” *input*

and *output* as *yes*. There might well be another yes/no question later on in the spreadsheet, and the bot would then get confused.

Blank inputs:
If you send the bot a blank input, it will return to the *output start* and restart the conversation. You can override this by providing an *output* labelled *blank_input*. If it is provided, it will be used instead of going back to the start.

Note that spaces and punctuation are stripped from the user input before it is processed. So typing a few spaces or … or just !?! is also considered a blank input. The input must contain at least one letter or number to be processed.

Diagram:
Below is a digram of this bot's structure. It shows how the *outputs* and *inputs* are connected together. Diagrams like this are useful for planning bots.



Important points:

- *inputs* following (in the spreadsheet) an *output start* are always active
- *inputs* following (in the spreadsheet) normal *outputs* are only active when the bot has just said that *output*
- the bot will favour *inputs* below *outputs* if it has a choice (when both *inputs* match their minimum accuracies)
- if two *inputs* produce the same score, the bot will choose the one with the higher accuracy; if the accuracies are also the same, it will choose the first one
- *inputs* without a *Goto* go to an *output* with the same label
- otherwise the *Label* should be unique
- provide an *output* labelled *blank_input* to handle blank inputs, or else the bot returns to the *output start*

Example 1.4 – One Line Input/Outputs

In November 2014 we added a new and simpler format for simple bots. You can now specify a *Type* of *inout*, and then put the *input* and *output* on the same line. The *input* text goes in the repurposed *Description* column, and the *output* text goes in the *Text* column.

Type	Label	Description/input text	Output text	If	Learn	Goto	Accuracy
output start	welcome	The first thing the bot will say.	Hi there, I'm your fourth basic bot.				
inout		Hello.	Hello back.				75
inout		How are you?	Fine thanks.				75
inout		Anything else.	Sorry, I don't understand.				0

As you can see *inout*s don't even need a *Label*, though you can provide one if you would like.

Over the next 60+ pages, you will learn many new and wonderful things, including how to add *phrases*, set variables, do maths, use wildcards, and much more. *Inout*s support all of these things.

In most respects, *Inouts* behave like *outputs*. They treat their *If*, *Learn*, *Goto* and *Mode* columns as an *output* does. Their only *input* like behaviours are the input text in the *Description* column and the *Accuracy.*

Important points:

- *inout* lines allow you to have inputs and outputs in the same line on the spreadsheet
- they use the *Description* column for the input text
- they have an *Accuracy* relating to the input text
- otherwise, they behave like an *output*
- they do not need a *Label*

Section 2 – Phrases

The peas bot is very fussy at first – it only accepts the word “hello” and minor variations thereof. After the peas question it gets very lax – accepting almost anything that starts with “y” or “n”. This section shows how to quickly and vastly expand the words that a bot will recognise.

Example 2.1 – Lots of Hellos

There are many ways to say “Hello” in English, such as “Hi” and “Hiya” and “Hi there”. We want the bot to recognise all of them. To do this, insert extra lines into your spreadsheet and add the different variations to the *Text* column. This example is based on the first or second bots we created.

Type	Label	Description	Text	If	Learn	Goto	Accuracy
output start	welcome	The first thing the bot will say.	Hi there, I'm your first bot of slightly greater understanding.				
input	hello	User says hello	hello			hello_back	60
			hi				
			hiya				
			hi there				
output	hello_back	Bot replies	Hello!				

This is a very simple bot. If it recognises one of those four greetings it will say “Hello!” or else it will go back to its starting *output*.

Note that the *Goto* and *Accuracy* refer to the *input* as a whole, not to the individual bits of text. The 60% applies to all four of the variations. So this will also recognise things like “hiy” because it matches 74% to “hiya”.

An alternative format for this is to put all the *Text* in a single row of the spreadsheet and separate the different variations with a /. This allows you to keep the entire *input* in a single row, but means the *Text* column might get wide and unreadable. You must put spaces around the /, so it doesn't get confused with a normal slash. Here's an example:

input	hello	User says hello	hello / hi / hiya / hi there			hello_back	60

Important points:

- the *Text* column can spread over several rows in your spreadsheet
- when using this technique, all other columns should be blank
- alternatively you can have several variations in the same row, separated by /

Example 2.2 – Introducing Phrases

This example introduces one of Cleverscript's Unique Selling Point – the main reason why it is so powerful. It extends upon the example above in a simple but elegant way. We call them phrases. Here is an example:

Type	Label	Description	Text	If	Learn	Goto	Accuracy
output start	welcome	The first thing the bot will say.	Hi there, I'm your second phrases bot!				
input	user_hello	User says hello	((hello))			hello_back	60
			((hello)) there				
output	hello_back	Bot replies	Hello!				
phrase	hello	My first phrase	hello				
			hi				
			hiya				
			hey				

Notice that the *input* hello only has two variations now: “((hello))” and “((hello)) there”. The double parentheses enclose a *phrase*. This *phrase* is specified at the bottom of the spreadsheet. It contains 4 variations: hello, hi, hiya, hey. Note also that I renamed the *input* to *user_hello* so that it wouldn't conflict with the *phrase* labelled *hello*.

The *input* line can now recognise a total of 8 different things: hello, hi, hiya, hey, hello there, hi there, hiya there, hey there.

Phrases are so powerful because they are multiplicative. In this case, 2 variations (“((hello))” and “((hello)) there”) times 4 variations (hello, hi, hiya, hey) makes 8 total possible variations. But this is the tip of an iceberg. Phrases can contain other phrases, and can even refer back to themselves. It's very easy to build up to millions of variations, and the bot will still process them quickly and efficiently. The next example shows is more filled out.

Important points:

- *inputs* can refer to *phrases*
- *phrases* are specified by putting the word “phrase” in the *Type* column
- *phrases* are referred to using double parentheses, such as ((hello))
- *phrases* multiply the number of variations that a bot can recognise

Example 2.3 – Rapidly Expanding Phrases

This is a longer example of phrases, showing how they multiply, and how they can be used for output as well.

Type	Label	Description	Text	If	Learn	Goto	Accuracy
output start	welcome	The first thing the bot will say.	Hi there, I'm your third phrases bot!				
input	asks_name	User asks for bot's name	((what is)) your name?				40
			((what are)) ((you)) ((called))?				
			Who are ((you))?				
output	asks_name	Bot replies	My name is ((bot_name)).				
			I am called ((bot_name)).				
			I am ((bot_name)).				
			((bot_name)).				
phrase	what is		((what)) is				
			((what))'s				
			((what))s				
phrase	what are		((what)) are				
			((what))'re				
			((what))re				
phrase	what		what / wot / wat / whaat				
phrase	you		you / u / ya				
phrase	called		called / named				100
phrase	bot_name		Evie				

Your bot will now respond to anything from “What is your name?” to “Wot're u called?” and “who are ya?”. In fact, it will accept 87 variations of input, along with anything else that is 40% similar to one of those 87 variations.

That is computed like this: The *input* labelled *asks_name* can accept 12 variations for “((what is)) your name?” made of up 3 variations for ((what is)) * 4 ((what)). Along with 72 for “((what are)) ((you)) ((called))?”: 3 variations of ((what are)) * 4 ((what)) * 3 ((you)) * 2 ((called)). And a further 3 for “who are ((you))?”

This example also introduces variations for the *output*. Unlike on *input*, case and punctuation are not ignored for *output*. The output *asks_name* has 4 variations, each referring to the *phrase* labelled *bot_name*. *bot_name* has no variants. It is just a placeholder. So if you wanted to change the bot's name to Linda or Bartholemew, you only need to make the change in one place.

With 4 variations to choose from, the bot's default behaviour is to choose the one most like the user's input. This sort of imitation helps to produce a more realistic conversation. So if you ask “What is your name?” it is likely to reply “My name is Evie.” If you ask “What are you called?”, the bot will reply “I am called Evie.” This behaviour can be modified, and will be explained in a

later example.

This example also lacks any *Gotos*. The *input* labelled *asks_name* maps directly to the *output* labelled *asks_name*. It uses an *Accuracy* of just 40% to give an even wider range of accepted inputs.

This example also shows the *Accuracy* being used on a *phrase* as well. The *phrase* labelled ((called)) must match 100%. So “what are you callled?” will not work but “what arre you called?” will. This is useful when some part of the user's input must match really well, such as a password. When the column is left empty, the default accuracy for *phrases* is 30%.

This bot is called Evie in honour of the bot at www.existor.com. That bot is powered by a script to answers questions about Existor (such as “what is your phone number?”) combined with some general chatting ability borrowed from www.cleverbot.com.

Important points:

- *phrases* can contain other *phrases*
- *phrases* can have an *Accuracy* requirement just like *inputs*
- *phrases* can also be used within *outputs*
- if there are multiple variations, the bot chooses the *output* closest to the *input*

Phrasing Guidelines

Phrases are very flexible. They can be used to represent and parse any language. So before embarking on a complex bot it is useful to have some guidelines and standards for labelling phrases. Below are our suggestions.

Synonyms

Phrases are easiest to use and understand as groups of synonyms. For example, the words “bike” and “bicycle” are almost always interchangeable. So it makes sense to have a phrase labelled ((bicycle)) containing: bicycle / bike. It's also useful to add common mis-spellings and typos like “bycycle”.

Synonyms in context

However, very few words are exact synonyms of each other. If phrases only contained synonyms and typos they wouldn't be very practical.

Usually words have overlapping meanings. For example, a “pet” is almost always an “animal”, but an animal is not always a pet. Some words are quite broad, a “boat” can be a yacht, ferry, ship, paddle steamer or rowboat. Others are more specific, like “yacht”.

So it depends on the context. If you are creating a bot for a fantasy adventure game, then ((boat)) can safely be quite broad. But on a website selling boat parts, it should be more specific.

This is the main difficulty in creating generic libraries of phrases which can be used by any bot. The breadth of the phrases depends on the context of the bot.

A rule of thumb is that the closer a phrase is to the bot's main purpose (a character in a fantasy game, or selling boat parts) the narrower it should be. Phrases less related to the bot's purpose can be much broader.

Most specific

Sometimes some words are unnecessary. For instance, you can say “I've got to go” or just “Got to go”. There are many statements like this where the “I've” or some other part of it is not always needed.

In these cases, your phrase should be as long and specific is possible. In other words, your phrase should be labelled ((I have got to go)) instead of ((got to go)). Then there is less confusion over what it actually means, and the longer phrase can contain the shorter phrase.

So the phrase ((I have got to go)) can contain: ((I have)) ((got to go)) / ((got to go)). It will work with or without the ((I have)). To complete the example, ((I have)) would contain: I have / I've. And ((got to go)) would contain: got to go / gotta go.

In other words, the phrase label is best as the most clearly expressed way of completely describing a meaning, which will often be a longer word or phrase, yet at the same time you want it to be a common-enough way of expressing that meaning so as to feel natural, and not stilted, when reading a complete sentence.

Types of things

A bot for an online pet shop might begin by asking “What type of pet do you have?” In this case, it is more logical for the corresponding input to be “I have a ((pet_type))” rather than just “I have a ((pet))”.

The ((pet_type)) phrase can then contain: dog / cat / horse / ferret / gerbil / camel / etc. The ((pet)) phrase should be reserved for words that are rough synonyms of the word “pet”, such as: pet / animal / creature.

We therefore recommend that phrases containing types or lists of things should end in _type. So ((colour_type)) could contain: blue / red / green / yellow / etc. Whereas ((colour)) contains: colour / color / hue / tint.

An alternative to _type is _list. We tend to use _list for lists of things the bot knows about. For example, an informational bot on a website may be able to talk about 20 things. We would put those 20 into a ((topic_list)) phrase. In the colours example, ((colour_type)) might contain a list of all (or lots of) colours, whereas ((colour_list)) contains just the colours that the bot can talk about.

Multiple Definitions
Many words have several distinct definitions. For example ((pet)) can be a noun as above, or a verb meaning “to stroke”. This usually doesn't cause an issue, as the context of the bot will determine which definition is in being used.

If you are creating a bot to go on a pet care website though, you might need to refer to both. In these cases, we recommend that the phrase ((pet)) refers to the most common usage, probably as a noun in this case: pet / animal / creature / etc.

Less common usages can be labelled with a suffix, such as ((pet_stroke)). A better idea however would be to have a phrase called ((stroke)) to cover that meaning.

Some words have dozens of definitions. According to an online dictionary, the word “set” has 119 distinct definitions, including 64 as a verb and 29 as a noun. Many of these are related, so the situation isn't quite that bad, but it does mean that there may not be a most common usage. Putting all the meanings of “set” into ((set)) wouldn't be right either. The purpose of phrases is to distinguish meaning not blur it. Either way (referring to the most common usage, or including all meanings) labelling the phrase as simply ((set)) would be confusing.

In these cases, you could put a suffix such as ((set_put)), ((set_fix)), ((set_decide)), ((set_tennis)). Though again it might be less confusing to use alternative labels: ((put)), ((fix)), ((decide upon)) and ((tennis set)). But ((put)) also has many definitions, so at some point suffixes will be necessary.

If you needed even more specificity, you could decide upon a dictionary to use, and then have phrase labels like ((set_noun1)) which means the first noun definition of the word “set”. Though going down this route will involve lots of looking up words.

Prepositions
Prepositions also have multiple definitions, and they can be synonyms for each other in different situations. For example, “I would like to talk with you”. The “with” here could be “with” or “to”.

To accommodate this, you can either use the multiple definition technique above, such as ((with_to)) which contains: with / to. Or ((to_with)) containing: to / with. Or you can include the ambiguity in a longer phrase and have ((talk with)) containing: ((talk)) with / ((talk)) to.

Fortunately, most prepositions are very small, so even if you ignore this issue entirely, your phrases will probably still work if the Accuracy is set low enough.

Bot preferences
You can also store your bot's preferences and properties in phrases. We recommend phrases labelled like ((bot_name)) to keep the bot's name and ((bot_colour)) to store their favourite colour. It is useful to have these as phrases so that if you need to change them, the change is only made in one place, as in the example above.

The prefix bot_ is needed as the phrase ((colour)) is for synonyms of "colour" as above. Prefixing them all with bot_ also helps to keep them together mentally and in the spreadsheet.

Remembering it all
As a bot grows, it can be difficult to remember what phrases you have used, especially if more than one person is involved in creating the bot. For example, you may not remember whether to use ((set_place)) or ((place_put)) in your phrases.

In this case, you can either:

- search your spreadsheet for previous usage (you can usually tick a box to "match entire cell contents" to make this easier)
- store all your phrases alphabetically so they are easy to look through
- create a new phrase anyway and see if there are warnings or errors when you upload (not really recommended)

The easiest method is probably to list your phrases alphabetically. To help with this, you could keep your phrases in a separate spreadsheet (separate from your inputs and outputs) and use only / as a separator (instead of multiple lines). Then you can easily add new phrases to the end and use the spreadsheet software's Sort facility. This is also very useful if you want to use the same set of phrases with several different bots.

Important points:

- *phrases* are like groups of synonyms
- how broad the groupings are depends on the context of the bot
- *phrases* closer to the context or goal of the bot generally have a narrower focus
- use the longest and most specific variation for your *phrase* label
- use _type for *phrases* that contain types of things, such as ((pet_type))
- when a word has multiple definitions add a suffix to differentiate, such as ((set_put))
- prefix bot properties and preferences with bot_, such as ((bot_name))
- put *phrases* in a separate spreadsheet, one on each line using / to separate the variations
- this allows *phrases* to be easily sorted to help in looking things up

Example 2.4 – Output Modes and Capitalisation

When your bot replies, it chooses the response most similar to the user's input. In the previous example, when you asked “what is your name?”, it replied “My name is Evie.” This *output Mode* is called “most like input”. There are six other *output Modes*: “random”, “random, same”, “random, different”, “in order”, “in order, repeat last” and “first”. To use this feature, you need to add a new column after *Accuracy* for the *Mode*.

Type	Label	D	Text	If	Learn	Goto	Acc	Mode
output start	welcome		Hi there, I'm your fourth phrases bot! I demonstrate modes and capitalisation.					
input	friendly		((hello))				30	
			((hello)) there					
output	friendly		((Hello)).					in order
			((Hello)) to you.					
phrase	hello		hi / hello / hey / hiya					random, different

When you greet this bot, it will cycle through the *output* labelled *reply* in order, first saying “((Hello))” and then “((Hello)) to you.” and then back to “((Hello))” etc. But it will output the ((Hello)) randomly. So if you keep saying “hello”, it will give replies like “Hi”, “Hey to you.”, “Hiya”, “Hi to you.”, etc.

The *output Mode* “random, same” gives a random output the first time and then says the same thing in subsequent interactions, and “random, different” gives random, but different each time until it uses up all the choices and then restarts. The *output Mode* called “in order, repeat last” is the same as “in order”, but it won't go back to the beginning. It will keep repeating the last one. If both the *Modes* above were “in order, repeat last”, then the bot would reply “Hi”, “Hello to you.”, “Hey to you” and then keep repeating “Hiya to you.” The mode “first” always outputs the first one.

The *output Mode* still works with *If* conditions, cycling through the matching ones as above. If none of the conditions match, it outputs nothing or the phrase *unhandled_ifs* (behaviour changed October 2014 to help debugging). You can also shorten the *Mode* to *m* for “most like input”, *r* for “random”, *s* for “random, same”, *d* for “random, different”, *i* for “in order”, *l* for “in order, repeat last” or *f* for “first”. If no *Mode* is provided, then “most like input” is used, as in the previous examples.

Notice the capital H in ((Hello)). It tells the bot to capitalise whatever word eventually gets output. This means that the same *phrase* can be output at the beginning or the middle of a sentence. If at the beginning, refer to it with ((Hello)). If in the middle, use ((hello)). At all other times, *phrase* labels are case-insensitive, meaning that upper and lower case don't really matter.

Important points:

- The *output Mode* goes in the new *Mode* column and applies to *outputs* and *phrases*
- It can be either: “most like input”, “random”, “random, same”, “random, different”, “in order”, “in order, repeat last” or “first”. If blank, the default is “most like input”
- You can shorten it to m, r, s, d, I, l or f.
- If all ifs are false, it returns a blank or the phrase *unhandled_ifs*
- All *Modes* are reset when the bot returns to its *output start*
- Capitalise the first letter of a *phrase* reference to force its output to be capitalised

Advanced Example 2.5 – Output Phrases

The next few examples discuss more advanced phrasing techniques. Feel free to skip to the Learning section and return here later.

The previous two examples demonstrated using *phrases* in *outputs*. This is an easy way to add variation to the *output*, so that the bot doesn't always say the same thing. It is also convenient to use the same *phrase* for *input* and *output*, as in the ((hello)) in the previous example.

However, there may arise a situation where you don't want to reuse the same *phrase* for *input* and *output*, but it would be confusing to give them separate labels. In this case, you can change the *Type* to *output phrase*. *Output phrases* are exactly like *phrases* but they are only used for *output*. Like this:

Type	Label	D	Text	If	Learn	Goto	Accuracy	Mode
output start	welcome		Hi there, I'm your fifth output phrases bot!					
input	friendly		((hello))			hello_you	30	
			((hello)) there					
output	hello_you		((Hello)).					in order
			((Hello)) to you.					
phrase	hello		hi / hello / hey / hiya					
output phrase	hello		hello					

This is exactly the same as the example above, except now the bot will only ever say “Hello” and “Hello to you”. It will still recognise “hi”, “hello”, “hey” and “hiya” but will be more formal in its reply.

Output phrases are also necessary if you want the same *phrase* label to have a *Learn* on *input*, but *Ifs* and *Learns* on *output*. Normally that wouldn't be allowed. If you tried that, then when the *phrase* was used for *input*, only the first part of the first *Learn* would be considered, and the *Ifs* and subsequent *Learns* would be ignored.

You can achieve the same effect but just having different *phrase* labels for your *input* and *output* phrases, but it feels more natural and logical to tie them together.

Important points:

- usually *phrases* can be used for both *input* and *output*
- *output phrases* are *phrases* that are only used for *output*
- you can have a *phrase* and an *output phrase* with the same label

Advanced Example 2.6 – Percentages within Phrases

You can use percentages within *phrases* to more subtly express how words relate to each other, how close the synonyms actually are. For example, the words “tree” and “shrub” are often interchangeable, as in “what a lovely tree” and “what a lovely shrub”. But not always, “monkeys live in trees” but not “monkeys live in shrubs”.

If you were creating a nature-loving bot, you might need to use the words “tree” and “shrub” a lot, so you'd probably want to put them in the same *phrase*. But you would still want to reserve the right to tell them apart once in a while. This example demonstrates how you can make a “shrubs” be 80% like “trees”.

Type	Label	Desc	Text	If	Learn	Goto	Accuracy
output start	welcome		Hi there, I'm your sixth phrases bot! I like percents.				
input	like_trees		Do you like ((trees))?				30
input	monkey_trees		Do monkeys live in ((trees))?				30
input	monkey_shrubs		Do monkeys live in ((shrubs))?				30
output	like_trees		Yes I love them about $accuracy$%.				
output	monkey_trees		Yes they do.				
output	monkey_shrubs		No they don't.				
phrase	trees		trees				
			80% shrubs				
phrase	shrubs		shrubs				
			80% trees				

The way to specify that “shrubs” are 80% like “trees” is to put the percentage in front of the word as above.

If you ask this bot “do you like trees?” it will reply “Yes I love them about 100%.” If you ask “do you like shrubs?” it will give the same reply, but with only about 98% accuracy, because the word “shrubs” only matched 80% but the rest of the sentence matched exactly. So in the context of whether the bot likes them, “trees” and “shrubs” are very similar.

But in the other context of monkey habitation habits, “trees” and “shrubs” are not interchangeable. So if you say “do monkeys live in trees?”, the *input* labelled *monkey_trees* will have a higher accuracy, and the bot will reply “Yes they do.”

What the percentages actually mean:

The actual percentages you assign are highly subjective, and only matter relatively. In other words, all that matters is that “shrubs” are less tree-like than “trees”. The exact number you put isn't very important.

However, it is possible to view it more logically. From a language perspective, we are saying that “shrubs” only belong 80% to the concept of “trees”. Imagine something this is only 1% like a tree, such as a blade of grass. It's a plant which grows upwards so has a tiny amount of tree-like properties. Then try to judge where a shrub fits on the scale of grass-to-trees – 80% seems about

right. This is still very subjective but now has some vaguely justifiable basis.

You could also approach it by trying to think of a word that is 0% like a tree, such as “interstellar” or “slowly”. These words are not even nouns. Then imagine where a shrub fits on the scale of interstellar-trees – maybe around 99.9%. Even a blade of grass might come in at 99%. In fact, doing it this way, all the things in the ((trees)) *phrase* would be in the high 90s. The problem with this approach is that it leaves far less room for differentiation. In fact, it becomes impossible because the percentage has to be a whole number without a decimal.

The actual calculation which Cleverscript uses is similar to the grass-to-trees scale. It assumes that everything in the ((trees)) *phrase* already shows some resemblance to trees. When it comes across a percentage it adjusts the final accuracy score by something like: (a fraction) * (percentage provided) * (length of matched word “shrubs”) / (length of whole sentence).

Important points:

- put a percentage in front of a *phrase* variation to express “belongingness”
- use percentages to distinguish between words that are similar but not exactly the same
- the actual percentage you use is subjective and relative
- but if you prefer something more concrete, imagine something that belongs only 1% to the *phrase* and use that as a scale to judge what each percentage should be

Advanced Example 2.7 – Optional Phrases and Text

You can make *phrases* optional by adding a ? to the *phrase* reference, such as ((?please)). You can also do this with normal text using single parentheses, such as (?thanks). In this case, "thanks" is not a *phrase*, and a *phrase* labelled *thanks* doesn't need to exist.

Type	Label	Desc	Text	If	Learn	Goto	Accuracy
output start	welcome		Hi there, I'm your seventh phrases bot! I demonstrate optionals.				
input	tell_me		((?please)) tell me something ((?please)) (?thanks)				30
output	tell_me		You are great.				
phrase	please		please				
			pretty please				
			would you ((please))				

You can be ultra polite to this bot. You can say "please tell me something would you pretty please thanks" and it would match 100%, as would just "tell me something".

Also notice that phrases can refer to themselves, "would you ((please))" refers back to itself which allows things like "would you pretty please". This is called recursion. A phrase is only allowed to refer back to itself once.

Internally, optional text is processed separately from optional phrases. Optional text is processed at the beginning and split into different *phrase* variations, which means that "(?please) tell me" has the same effect as "please tell me / tell me". It also means that you can put *phrases* and wildcards (see future example) within optional text, such as "(?would you ((please)))". But you can not put *phrases* and wildcards inside *phrases*, so "((?would you ((please))))" is NOT supported. To achieve that, you would need a ((?would you)) *phrase* containing "would you ((please))".

Important points:

- Make a *phrase* optional by putting a ? inside the parentheses, such as ((?please))
- Make text optional using single parentheses, such as (?thanks)
- *phrases* can refer back to themselves

Advanced Example 2.8 – Unimportant Phrases

In the last example the ((?please)) is optional but it still affects the overall *Accuracy*. For example, if you type “tell moi something”, it would match “tell me something” with 77%. But “please tell moi something” matches “please tell me something” with 84%. And “pleeeease tell moi something” matches 69%. You may want these all to have the same score, as if the “please” was never there at all.

You can do this by adding an ! to the phrase label such as ((!?please)). It works for non-optional phrases too such as ((!please)).

Type	Label	Desc	Text	If	Learn	Goto	Accuracy
output start	welcome		Hi there, I'm your eighth phrases bot! I show unimportant things.				
input	tell_me		((!please)) tell me something ((!?please))				60
output	tell_me		You are fantastic.				
phrase	please		please				
			pretty please				

In this example, the first ((!please)) is required but the second one is optional. And neither contribute to the *Accuracy*. Both “please”s will be stripped off your input before the final *Accuracy* is computed.

Unlike optional phrases using a ?, unimportant phrases with a ! can only occur within *phrase* labels. They can not apply to normal text, so (!please) would not work.

This example requires a minimum *Accuracy* of 60%. That is 60% *before and after* the “please”s are stripped off. So if you type “please tell moi sumthing”, it first compares it to “please tell me something”. This matches about 62% so it passes. Then it strips off the “please”s and compares “tell moi sumthing” to “tell me something” which is only about 50%. This is the final score, and as it is below 60%, this *input* doesn't match.

Important points:

- Make a *phrase* unimportant by putting a ! inside the parentheses, such as ((!please))
- Make a *phrase* optional and unimportant by putting ! and ?, such as ((!?please))
- The reverse ((?!please)) also works
- The *Accuracy* is checked before and after the unimportant *phrases* are removed

Advanced Example 2.9 – Wildcards

The *Accuracy* column can be used to add fuzziness to your bot. If you try to match “hello” with a low accuracy, it will probably match “heya” and “hellllo” as well.

Another way to add fuzziness is using wildcards. They are like the joker in poker, they can be whatever you want. There are two types of wildcard in Cleverscript. An underscore _ matches any single character. And a squiggly tilde ~ matches zero or more of anything.

For example, the user input “hello” will match “he__o” with nearly 100% accuracy, because the _ wildcards are assumed to be letter Ls. Similarly it will match “h~” with nearly 100% as the ~ eats up the “ello” part. The accuracy is less than 100% because wildcard matches are never considered as good as true matches like “hello”=“hello”.

Wildcards, combined with *$variable_other$* provide a powerful way to learn things about the user, as you can see in this example:

Type	Label	Desc	Text	If	Learn	Goto	Accuracy
output start	welcome		Hi there, I'm your ninth phrases bot! I'm wild. Guess my favourite colour.				
input	right_colour		~ ((bot_colour))				90
input	anything_else		Anything else				0
output	right_colour		Yes! Well done with accuracy $accuracy$%!				
output	anything_else		Sorry. Try again.				
phrase	bot_colour		green				

This bot asks the user to guess it's favourite colour. Without wildcards, you would have had to predict all possible ways of saying “I think that your favourite colour is...”. Now, as long as the user's input end with the word “green”, it passes.

This is very useful if you are looking for a specific word, as in this example. However, in bots with lots of phrases, it should be used with care, as it can often pick up unexpected inputs.

For example, if your prediction was “~ red”, this would match fairly well to anything containing the letters “red” such as “altered”. Pretend you had another prediction containing the word “altered” such as “my phone number is altered”. If the user typed “my num is altered”, this would match better to “~ red” than “my phone number is altered”. This is a very contrived example, but becomes more likely as a bot grows in size and complexity, especially if you have more than one ~ on a single line.

Important points:

- Use ~ as a wildcard, it can match anything the user says
- Use _ as a mini-wildcard, it can match any single character
- Use ~ and _ sparingly as they can cause unexpected results
- Wildcard matches can never be 100%

Section 3 – Learning

With just *inputs*, *outputs* and *phrases*, you can already create complex bots which respond to user input and give useful information. However, Cleverscript can also learn data, and you can use that data to streamline your script and find out things about the user.

Example 3.1 – Basic Learning

You may remember learning about variables in your first algebra class. Given an algebraic equation like *x+4=10*, the variable called *x* has the value *6*. Computer programming languages are also full of variables, though the variables are often words rather than single letters, things like *counter=1*.

The *Learn* column of your spreadsheet supports a similar format. Any text for an *output* or *phrase* can have an equation like *friendly=yes*. Here the variable is called *friendly* and the value is *yes*. When that *output* or *phrase* is used, the variable is learned. Here's a simple example:

Type	Label	Description	Text	If	Learn	Goto	Accuracy
output start	welcome	The first thing the bot will say.	Hi there, I'm your first learning bot!				
input	hello	User says hello	((hello))			thanks	60
output	thanks	Bot's reply	Hmm, thanks.				
phrase	hello	Hello phrase	hi		friendly=yes		
			hello		friendly=maybe		
			go away		friendly=no		

When you upload and run this example, the bot will deliver its opening line as usual. If you say "hi", the *input* labelled *hello* will match it via the phrase *((hello))*. Notice that the *Learn* column next to the word "hi" says *friendly=yes*. When you say "hi", the variable *friendly* will be set to the value *yes*. Likewise, if you say "go away", *friendly* will learn the value *no*. For all these, the bot will reply "Hmm, thanks".

If you test this using the uploader, it will tell you the values of any variables learned, along with their age, such as: *friendly=yes, friendly_age=0*. This means that the variable *friendly* now has the value *yes*, and this value is *0* interactions old (it was just learned). The next two examples extend upon this, and then you'll find out how to actually use them.

If you say anything else, the bot will return to its *output start* as normal. This will clear all variables, so *friendly* will go back to being blank.

Inputs can also learn things, but their *Learns* are associated with the *Goto* column instead and are discussed in a later section of this manual. Note that if you wanted to put the "hi / hello / go away" on one line in the spreadsheet with / in between, you would have to do the same with the *Learn*: "friendly=yes / friendly=maybe / friendly=no".

Important points:

- *outputs*, *output starts* and *phrases* can learn variables in the *Learn* column
- use the format *variable=value*
- only letters, numbers and _ are allowed in variable names, no spaces
- when the bot returns to its *output start*, all variables are cleared

Example 3.2 – Learning From Input

You don't have to specify a value when learning in *phrases*. This will cause your bot to learn directly from the user's input.

Type	Label	Description	Text	If	Learn	Goto	Accuracy
output start	welcome	The first thing the bot will say.	Hi there, I'm your first learning bot!				
input	hello	User says hello	((hello))			thanks	60
output	thanks	Bot's reply	Hmm, thanks.				
phrase	hello	Hello phrase	hi		greeting		
			hello		greeting		
			go away->bad		greeting		

Note that the variable *greeting* does not have an equals sign. Instead, it will learn whatever is in the *Text* column. For the "hello", it will learn the value "hello", which is functinoally exactly the same as putting *greeting=hello,* but is less data entry (especially when combined with a short-cut shown later in this section).

It will also learn the actual user's input into *greeting_other*. So if you make a typo and say "helllo", *greeting* will be set to "hello" (from the *Text* column) but *greeting_other* will be "helllo" (the actual input with the typo). Along with *greeting_age*, you will see these values when you upload and test your bot. This feature becomes very useful later in this section when wildcards are introduced.

The -> in the last row above overrides learning "go away" into the variable *greeting*. Instead it will learn "bad". However *greeting_other* will still learn the actual user's input with misspellings and all "go away" or "go awway".

This kind of learning can also be done within *outputs* and *phrases* during output.

Important points:

- *phrases* can learn parts of the matched input
- to do this, put a variable name by itself in the *Learn* column
- all variables are cleared when the bot returns to its *output start*

Example 3.3 – Using What You've Learned

There are two main ways to use the variables the bot has learned. The first method is to put the value directly into an *output* (the other method is discussed in Section 4). To output a variable you need to surround it by $ signs. The example below learns a value for the variable *greeting* and then outputs it with *$greeting$* when the bot replies. It also outputs the internal Cleverscript called *accuracy*. There are several internal variables listed after this example.

Type	Label	Description	Text	If	Learn	Goto	Accuracy
output start	welcome	The first thing the bot will say.	Hi there, I'm your third learning bot! I output a variable.				
input	friendly	User is friendly	((hello)) / ((hello)) there			thanks	30
output	thanks	Bot's reply	Thanks for saying $greeting$ with accuracy $accuracy$%.				
phrase	hello		hi		greeting		
			hello		greeting		
			hiya		greeting		

If you say “hello” to this bot, it will reply “Thanks for saying hello with accuracy 100%.” It has learned and output the value of the variable *greeting* along with the internal variable *accuracy*.

You can also output a variable's age and the actual user input which matched, using the format *$greeting_age$* and *$greeting_other$*. The age is the number of interactions since the variable was last learned. Several other properties are also available. If you typed “helllo” above then *greeting=hello, greeting_other=helllo*, *greeting_age=0*, *greeting_length=5* (the number of characters in *$greeting$*), *greeting_other_length=6* and *greeting_position=2* (as “hello” is the second choice in the *phrase* labelled *hello*).

You can also use variables in *inputs* and *phrases*. This allows you to change your predictions depending on what the user has said previously. This is useful for things like asking a user to repeat something (like an email address or password).

Important points:

- output variables by putting them in $ signs, like this *$variable$*
- output a variable's age with *$variable_age$*
- output the actual matched user input with *$variable_other$*
- output a variable's length *$variable_length$* and *$variable_other_length$*
- output the variable's position within the phrase with *$variable_position$*
- capitalise variables just like phrases with *$Variable$*
- you can also use variables in *inputs* and *phrases*

Internal Variables

Every bot manages several internal variables. You can output their values whenever you would like:

- *interaction_count*: how many pairs of bot/user interactions have occurred so far
- *input*: the entire user input, with any spaces trimmed off both ends
- *input_label:* the label for the *input* which matched the user's input
- *input_id*: the internal id number corresponding to the *input_label*
- *filtered_input: user input after input filtering*
- *predicted_input*: the scripted text which the user input matches
- *accuracy*: the percentage match between the user's input and the *input* row
- *output_label*: the label for the *output* which the bot is saying right now
- *output_id*: the internal id number corresponding the *output_label*
- *output*: the actual thing the bot is saying right now
- *conversation_id*: identifier for this conversation between user and bot
- *errorline*: any error information
- *database_version*: version of the database
- *software_version*: version of the software
- *time_taken*: the number of milliseconds the bot took to respond
- *random_number*: random number from 1 to 1000 reset every interaction
- *time_second*: number of seconds on the system clock (0-59)
- *time_minute*: number of minutes on the system clock (0-59)
- *time_hour*: number of hours on the system clock (0-23)
- *time_day_of_week*: current day of the week (0-6 for Sunday to Saturday)
- *time_month*: current month number (1-12 for January to December)
- *time_year*: current four digit year (2013)
- *time_started*: number of seconds from 1 Jan 1970 to when conversation started
- *time_elapsed*: approximate number of seconds since conversation started
- *interaction_1* to *interaction_20*: record of the previous twenty interactions
- *callback*: used by some versions of the software to store a callback function

For instance, in the example above, if you say “hello”, then *input=hello*, *predictedinput=hello*, *input_label=friendly*, *accuracy=100* and *output_label=thanks*. The random number and time variables are set when the bot starts processing the user's input.

It can be very useful to set up a *phrase* labelled *((debug))* in your bot, which outputs user and internal variables along with every *output*. This will be demonstrated in a later example.

In addition to the above, the following variables relate to the Clever Data feature, which allows your bot to do some general small talk. See section 6 for more information on how this works.

- *reaction*: reaction returned by Clever Data
- *reaction_tone*: tone of the reaction from very negative (-2) to very positive (2)
- *reaction_degree*: percentage estimate of how much this reaction is felt
- *reaction_values*: 7 comma-separated percentage values, see below
- *emotion*: emotion returned by Clever Data
- *emotion_tone*: tone of the emotion from very negative (-2) to very positive (2)
- *emotion_degree*: percentage estimate of how much this emotion is felt
- *emotion_values*: 7 comma-separated percentage values, referring to 7 basic emotions (anger, fear, disgust, contempt, joy, sadness, surprise)
- *clever_match*: used to influence Clever Data match
- *clever_accuracy*: accuracy of match made by Clever Data
- *clever_output*: reply returned by Clever Data

Advanced Example 3.4 – Learning Multiply and Mathematically

It is very easy to learn more than one variable, and to perform basic mathematical operations on variables. And from November 2013 Cleverscript also includes a more advanced mathematical expression parser, demonstrated and described below.

Type	Label	Desc.	Text	If	Learn	Goto	Acc
output start	welcome		Hi there, I'm your fourth learning bot!		mood=okay and count=1		
input	greeting		((hello))			thanks	60
output	thanks		You've greeted me $count$ times. I'm in a $mood$ mood.		count=+1 and test=(3.14+$count$/2)		
phrase	hello		hello		greeting and mood=good		
			go away		greeting and mood=bad		

When a bot first starts chatting, all its variables are blank. The example above uses the *output start* to assign default (aka starting or initial) values to the *mood* and *count* variables. After this bot says "Hi there!", it learns the variable *mood* as *okay* and *count* as *1*. The word "and" tells it to do a multiple learn.

If you say "hello" or "go away" you will trigger the *Learn* in the *phrase* labelled *hello*. This will also learn two variables. The variable *greeting* does not have an = sign, so it will learn from the *Text* column and the user's input. The variable *mood* will learn either *good* or *bad* depending on what you said.

Finally, after the bot delivers the *output* labelled *thanks* it will increment the variable *count*, adding 1 to its value each time. You can use this syntax to add, subtract, multiply, divide or modulus any whole number. It ignores and truncates everything after the decimal point. Unless you specify a default value in the *output start* all numerical variables start at 0.

Mathematical expressions:

You can also use mathematical expressions when learning (and in *If* conditions – see next section). Math expressions give you the ability to manipulate decimal numbers in much more complex ways. The expressions:

- should be enclosed by parentheses
- can include whole numbers, decimals, +, -, *, / and %
- as is standard, + and – have lower precedence than other operators
- can have more parentheses inside them to determine precedence
- spacing within the expressions is ignored

The following functions are also available:

- the *round* function rounds a decimal to the nearest whole number, eg *test=(round($test$))*
- the *floor* function always rounds down, so 9.99 becomes 9
- the *ceil* function always rounds up, so 9.01 becomes 10

Please note the following restrictions and warnings:

- don't put spaces around a division sign, or else it will be interpreted as a line separator, so put e.g. $test$/2 (without spaces) instead of $test$ / 2 (with spaces).
- if you use the operators above (+=, *=, etc) values are converted back to whole numbers, so put the + within the expression such as: test=($test$+1.5)
- all variables used in math expressions should be numbers, expect blanks which become 0
- if they are not numbers, then the expression will abort and the value will be 0
- division and modulus by 0 will not give an error, but will be evaluated as 0
- it will try to detect problems when uploading and display an error

An example expression is included in the example above but they can be used simply to perform operations on numbers with decimals (as += only works on whole numbers), or can be much longer and more complicated:

- test = ($test$ + 3.1428)
- test = ($count$ * ($test$ + 7) – (floor ($othervariable$) + 3) / 6.2)
- test = (round ($test$ + 17 – 3.18 / $count$))

Importing points:

- you can also *Learn* in *outputs* and *output starts*
- *Learns* in *output starts* essentially set default values for your variables
- separate *Learns* can be associated with each line of *Text*, unlike *inputs* and *phrases*
- you can do multiple *Learns* in *outputs, phrases* and *inputs* using the word “and” or the symbol &
- you can use =+ and =- to increment or decrement a numerical variable
- you can use =*, =/ and =% to multiply, divide and take the modulus of whole numbers only
- numerical variables start at 0
- mathematical expressions deal with decimals and must be enclosed in parentheses

Advanced Example 3.5 – Learning Shortcut in Phrases

Phrases and *outputs* support an additional shortcut for learning. If only the first *Learn* in a *phrase* is filled in with a single variable, it is assumed to apply to every line of *Text* in the *phrase*. This example is very similar to the second one above:

Type	Label	Description	Text	If	Learn	Goto	Accuracy
output start	welcome	The first thing the bot will say.	Hi there, I'm your fifth learning bot!				
input	hello	User says hello	((hello))			thanks	60
output	thanks	Bot's reply	Hmm, thanks.				
phrase	hello	Hello phrase	hi / hiya		greeting		
			hello / hey				
			go away->bad				

The main difference is that all 5 of the *Text* options for the *phrase* labelled *hello* will automatically learn into the variable *greeting*, including those separated by the /. This can reduce the amount of data entered into a spreadsheet.

You can also provide a value such as *friendly=yes*. It will still be applied to all the lines of *Text* as long as it appears first and is the only *Learn*. As soon as you add a *Learn* next to any of the other lines of *Text*, or try to put a multiple *Learn* into the first line, then this short-cut will no longer work and each *Text* will learn separately.

This can cause ambiguity. If you only wanted to learn *greeting* for the first word “hi” above and not for any of the others, then you might be surprised that *greeting* was also learned for “hiya”, “hello” and all the others. To get around this, you can change it into a multiple *Learn* such as “greeting and test=yes” or swap them around so that “hi” does not appear first.

Importing points:

- a single *Learn* next to the first line of *Text* in an *output* or *phrase* will be applied to all *Text*
- this provides a data entry short-cut but there is a chance for ambiguity

Advanced Example 3.6 – Calculating and Overloading

This example shows how to deal with numbers, and an advanced technique for using the same phrase with two different variables.

Type	Label	D	Text	If	Learn	G	Acc
output start	welcome		Hi there, I'm your sixth learning bot! Let me calculate.				
input	calculation		what is ((num1)) ((operator)) ((num))	operator="+"	num1+=num		50
				operator="-"	num1-=num		
				operator="*"	num1*=num		
output	calculation		The answer is $num1$.				
phrase	digit		0 / 1 / 2 / 3 / 4 / 5 / 6 / 7 / 8 / 9				
phrase	num		((digit)) / ((digit))((digit)) / ((digit))((digit))((digit)) / one->1 / two->2		num		
phrase	num1		((num))		num1=num		
phrase	operator		+ / - / * / plus->+		operator		

The ((num)) *phrase* can match to any three digit number. It does this using the ((digit)) phrase. When you give this bot an input such as "what is 23+12", the "23" matches the *phrase* ((num1)), which contains ((num)). ((num)) learns the variable *num* as "23" and then *num1* learns the value of *num*, "23". The second number "12" matches the *phrase* ((num)) and is learned as *num*.

Depending on the *operator* variable, the variable *num1* (23) then adds, subtracts or multiplies the variable *num* (12) leaving the result in *num1*, which is displayed as the answer.

With a bit of extra scripting, this bot also recognises text inputs like "what is two plus one?". Also note that the spaces between ((num1)) and ((operator)) and ((num)) are optional. In fact, all single spaces in Cleverscript are optional. This allows it to match more fuzzily.

Importing points:

- to match numbers, implement a ((digit)) *phrase* with 0 / 1 / 2 / 3 / 4 / 5 / 6 / 7 / 8 / 9
- overload a *phrase* by wrapping it in another *phrase*, allowing the same *phrase* to learn to more than one *variable*
- all single spaces between phrases are considered optional

Advanced Example 3.7 – Learning with Wildcards

Wildcards, combined with *$variable_other$* provide a powerful way to learn things about the user, as you can see in this example:

Type	Label	Desc	Text	If	Learn	Goto	Accuracy
output start	welcome		Hi there, I'm your seventh learning bot! What's your name?				
input	users_name		((my_name))				80
			((!my name is)) ((my_name))				
output	users_name		Thank you $name_other$.				
phrase	my name is		I am / I'm / Im				
			my name is / my name's				
phrase	my_name		~		name		

Up till now, the only way to find out the user's name would be to list every possible name in the *phrase* labelled *my_name*, which would be a bit impractical. But now, if you type “My name is Julie”, the wildcard ~ will match the word “Julie” and *$name_other$* will contain the actual matched text: “Julie”. So the bot will reply “Thank you Julie.”

Alternatively, you can just type “Julie” and the bot will learn *$name_other$* as “Julie”. However if you type “I refuse to tell you”, then the bot will learn that as *$name_other$* and will reply “Thank you I refuse to tell you.” This shows the power and danger of wildcards. They are powerful as they can match anything, and dangerous as they may match too much. Well – they're not that dangerous. You're not going to lose a limb or anything, but your bot may give some odd replies.

You can use more than one ~ in a single line, but the more _ and ~ you use, the more likely the bot is to misinterpret the input.

Notice the ! in the the phrase ((!my name is)). This makes it unimportant and excluded from the overall result. This means that if you type “my namme is Julie”, the typo in “namme” won't effect the overall result and the variable *name_other* will be extracted correctly as “Julie”. If you leave out the ! then the whole sentence would match the ~ with a higher accuracy and *name_other* would be learned as “my namme is Julie”.

Important points:

- Use ~ as a wildcard, it can match anything the user says
- Store the values in variables and access them with *$variable_other$*

Advanced Example 3.8 – Dynamic Variables, Arrays and Temporary Variables

Cleverscript also allows for dynamic variables. These are essentially variables within variables. You can use them to create arrays. For example:

Type	Label	D	Text	If	Learn	Goto	Acc
output start	welcome		Hi there, I'm your eighth learning bot! I do arrays.				
input	users_name		my name is ((my_name))		count+=1 and facts{count}=name		70
input	users_city		I live in ((my_city)) / my city is ((my_city))		count+=1 and facts{count}=city		70
input	what_is		What is my ((attribute))?				70
output	users_name		You are $user_name_other$.				
output	users_city		You live in $user_city_other$.		where@=yes		
output	what_is		Your $attribute$ is $user_{attribute}_other$.				
phrase	my_name		~		user_name		
phrase	my_city		~		user_city		
phrase	attribute		city / name		attribute		

The first two inputs learn the user's name and city into the variables *$user_name$* and *$user_age$*. These are normal everyday variables. However, the *output* labelled *what_is* references them using a dynamic variable *$user_{attribute}$*. This embeds the variable *$attribute$* at the end. So if you say "What is my city?", your bot will replace the *{attribute}* inside *$user_{attribute}$* with "city" and output the variable *$user_city$*.

Meanwhile the variable *$count$* is incremented each time you tell it something, and a facts pseudo-array is created. If you tell it your city then your name, then it will learn *$fact1$* as "city" and *$fact2$* as "name".

Dynamic variables add a powerful new feature to Cleverscript. They are however less efficient to process than normal static variables, so should only be used if needed.

From October 2013, Cleverscript also supports temporary learning, where a variable is learned for the current interaction only and then reset to its previous value. This is accomplished by putting an @ symbol before the = in a learn. In the above example, the variable *$where$* is set to "yes" when the bot says "You live in ….". By the start of the next interaction *$where$* will have been reset to its previous value (blank). Note that *$where_age$* is not reverted. The age always reflects when the variable was last set, whether temporary or permanent. The @ can be used with =+ or by itself when learning in phrases (eg by just putting *attribute@* in the last line above).

Important points:

- Use the syntax *$user{attribute}$* to create dynamic variables (variables within variables)
- These can be used to create and reference arrays
- You can use more than one such as *$user{counter}{attribute}$*
- Dynamic variables are not as efficient as normal variables
- Put an @ symbol before the = in a learn to make it temporary, lasting one interaction only

Advanced Example 3.9 – Parameters

From November 2013, a new feature allows you to pass parameters into *phrases*. If you are a programmer, then they will remind you of arguments to functions. They are used like this:

Type	Label	D	Text	If	Learn	Goto	Acc
output start	welcome		Hi there, I'm your ninth learning bot. I do parameters.				
input	what_place		((what *noun* do you *verb*, place, like))				70
input	what_food		((what *noun* do you *verb*, food, hate))				70
output	what_place		I like Antarctica.				
output	what_food		I don't like fennel.				
phrase	what *noun* do you *verb*		((what)) *noun* do you ((*verb*)) / tell me the *noun* you ((*verb*))				
phrase	what		what / which				
phrase	like		like / love / prefer				
phrase	hate		hate / dislike / detest				

The *phrase* labelled *what *noun* do you *verb** takes two parameters: *noun* and *verb*. These parameters are declared by putting them in *asterisks*. When this *phrase* is used in the *input* labelled *what_place*, the parameter *noun* is set to the word "place" and *verb* is set to the word "like".

Within the *phrase* itself, **noun** and **verb** behave just like variables, with the values "place" and "like". So the *phrase* will actually try to match "((what)) place do you ((like)) / tell me the place you ((like))". After this bot says "hi there..." you can say things like "what place to you prefer?" or "tell me the food you detest".

Note that your bot can still have separate variables called $noun$ and $verb$. The parameters *noun* and *verb* are only active within the *phrase*.

Internally, the parameters feature relies on variables and learning, and dynamically referenced *phrases* such as (($topic$)) or ((*verb*)) are not as efficient as static *phrases* like ((food)) or ((like)). Other restrictions are that they can not be used recursively (a parameterised *phrase* can not refer to itself) and can not be dynamic, such as ((var *parameter*, value)).

Important points:

- Use *asterisks* in the *phrase* label to create a *phrase* which has parameters
- Pass the parameters to the *phrase* by adding a comma and then the value such as ((phrase with *parameter1* and *parameter2*, value1, value2))
- Within the *phrase* the parameters are replaced by their values
- *Phrases* with parameters are not as efficient as *phrases* without them
- *Phrases* with parameters can not refer to themselves and can not be dynamic

Advanced Example 3.A – Learning From Phrases

From October 2014, this feature allows you to learn a variable from a phrase. This is an exciting addition, allowing you much more flexibility with learns.

By the way, I have called it example 3.A because if I number it 3.10 then the examples list on the chatting page of the Cleverscript website will go out of order (3.1, 3.10, 3.2, 3.3, etc). So these examples are numbered hexidecimally because A=10 in hex.

Type	Label	D	Text	If	Learn	Goto	Acc	Mode
output start	welcome		Hi there, I'm your tenth learning bot. I learn from phrases.		mood="((somewhat)) ((mood))"			
input	hello		hello				70	
output	hello		I am in a $mood$ mood.				70	
phrase	somewhat		sort of / somewhat					random
phrase	mood		good / great / tolerable / bad					random

The *$mood$* variable in the *output start* is learned when the bot gives its welcoming statement. The phrases *((somewhat))* and *((mood))* are expanded and the variable is assigned the result. So it could be something like *"sort of good"*. After you say "hello" the bot will reply with its mood. It will relearn a new random mood every time you go back to the *output start*.

The phrases must be in quotes, otherwise they are mistaken for a mathematical expression. Note that the full functionality of phrases is not available here. For example, the phrases are not correlated against the input, so the *Mode* "most like input" has no effect. And the phrases used can have *Ifs* but any *Learns* are not executed.

Important points:

- Phrases can now be used in the values for *Learns*
- The phrases must be in double quotes
- The phrases are treated as *output phrases*, and so can have modes
- You can use any mode except "most like input"
- Within the phrases, *Ifs* are checked but no *Learns* are executed

Section 4 – If Conditions

The learning section of this manual showed how variables like *$accuracy$* can be included directly in an *output*. The other main way to use variables is in *If* conditions. This section will show how the *If* column is used.

Example 4.1 – Conditional Outputs

In *outputs*, you can use the value of a variable to choose a specific output variation. In the example below, the entries in the *If* column relate to individual rows of *output*. If you say either "hey" or "hey there", then the variable *greeting* will learn the value *hey*, and this will cause the bot to choose the *output* "What's up?"

In this example, shading is used to indicate that the *If* column is tied to the *Text* column for *outputs*. It is also tied to the *Learn* column. Learning in *outputs* was introduced in the previous section and will be revisited in the summary at the end of the manual.

Type	Label	Desc	Text	If	Learn	Goto	Accuracy
output start	welcome		Hi there, I'm your first conditional bot!				
input	friendly		((hello))			greet	30
			((hello)) there				
output	greet		Hi!	greeting=hello			
			Hello there.	greeting=hi			
			What's up?	greeting=hey			
phrase	hello		hi / hello / hey		greeting		

The *If* condition *greeting=hi* tells the bot to only respond with the corresponding *output* text "hello there", if the variable *greeting* has the value *hi*. This could have also been done with multiple *input* and *output* lines, but you would have needed up to six *input* lines and three *output* lines.

The *If* condition can be much more sophisticated than this, checking for multiple variables and allowing for >, < and other types of check as well. That will be covered in the next example.

Note that the / format is still supported as a separator for both the *Text* and *If* columns. In this example, you could have put "Hi! / Hello there. / What's up?" as the *Text*, and "greeting=hello / greeting=hi / greeting=hey" as the *If*. The bot will correctly associate the first condition with the first text and so on.

These *If* conditions also work in *phrases* and *output phrases*, but not in *inputs* as they use their *Ifs* for choosing a *Goto* instead.

Important points:

- *outputs* and can have conditions in the *If* column
- there can be one *if* for each separate bit of text
- conditions usually have the format *variable=value*
- the corresponding text will only be output if the condition is met
- you can still use / to separate the *Text* and *If*
- *If* conditions also work this way in *phrases* and *output phrases*

Example 4.2 – Debugging and Blanks

You can use conditional outputs to show the values of variables while you are chatting, which is very useful for tracing issues and errors. The bot in this example is very similar to one of the previous ones, but with debugging built in.

Type	Label	D.	Text	If	Learn	Goto	Acc
output start	welcome		Hi there, I'm your second conditional bot, with debugging.				
input	friendly		((hello))		friendly=yes	thanks	30
			((hello)) there				
input	debug on		turn debug on		debug=on	thanks	100
input	debug off		turn debug off		debug=off	thanks	100
output	thanks		Thanks. ((debug))				
phrase	hello		hi / hello / hey		greeting		
phrase	debug		Friendly: $friendly$ ($friendly_age$). Greeting: $greeting$. Accuracy: $accuracy$. Input: $input_label$.	debug=on			
			[BLANK]				

You can say “hello” and “hey there” to this bot, and it will learn the *greeting* variable as before and say “Thanks”. Immediately after the word “Thanks” it will also output the ((debug)) *phrase*. It will look at both the output options for *debug*. The first one can only be output when the variable *debug=on*. Since all variables start out blank, this is false, so the bot will instead output the second option, which is a blank.

The special word [BLANK] was created for just this situation. It allows you to output nothing. Empty cells would otherwise be ignored, and sometimes spreadsheet software automatically clears cells which contain only spaces.

You can then say “turn debug on” which sets the variable *debug* to the value *on*. The next time you say “hello” and the bot says “Thanks”, it will again include the ((debug)) *phrase*. But this time the *If* condition *debug=on* will be true, and so it will output the long sentence containing all the variables. You can say “turn debug off” to turn off debugging.

Note that the debugging *inputs* have an *Accuracy* of 100%, so you have to type them exactly. If you type anything else like “boo”, the bot will return to the *output start* and reset all variables, including clearing *debug*. This type of debugging is very useful when testing your bot using an API or some other interface which doesn't show variable values.

Important points:

- Use learning and an *If* to implement some basic debugging, showing variable values
- Use [BLANK] to output nothing

Example 4.3 – Complex Conditions

Conditional *outputs* can be much more complex than the examples given above. They can also appear in *phrases*. Like this:

Type	Label	D.	Text	If	Learn	Goto	Acc
output start	welcome		Hi there! I'm your third conditional bot.				
input	friendly		((hello))		friendly=yes	mood	60
			((hello)) there				
input	unfriendly		((go away))		friendly=no	mood	60
output	mood		You are ((user_mood)).				
phrase	hello		hello->3		formality		
			hi->2				
			hey->1				
phrase	go away		be gone->3		formality		
			go away->2				
			scram->1				
phrase	user_mood		very nice	formality<=2 and friendly=yes			
			nice	formality>=2 and friendly=yes			
			a bit mean	formality=3 and friendly=no			
			nasty	formality<=2 and friendly~n			

This bot accepts six types of greeting: “hello”, “hi”, “hey”, “be gone”, “go away” and “scram” and uses them to set two variables: *friendly* and *formality*. It then replies with something like “You are very nice.” depending on those variables. The very last condition checks if *friendly* contains an “n”.

This introduces a few new features:

- *Phrases* can have *If* as well, but they are only used within *outputs* when the bot says something. They are ignored for *inputs*.
- Numerical conditions: if you use <, >, <= or >= then your *variables* will be treated as numbers and compared numerically
- You can use # for not equals: *friendly#yes*. Imagine the # symbol is like = crossed out.
- The ~ symbol means “contains”. It is case-insensitive. Use !~ for “not contains”.
- Multiple conditions: use the words *and* and *or* to specify multiple conditions (alternatively & and |). If you mix *ands* with *ors*, then the leftmost one takes precedence. In other words, the condition *friendly=yes and formality=1 or formality=2* is actually treated like *(friendly=yes and formality=1) or formality=2* which is probably not the intention. But you can rearrange it to make it work: *formality=1 or formality=2 and friendly=yes*.
- Put the value in quote marks if it contains spaces, such as: *friendly=“very nice”*
- If more than one *If* condition is met, then the bot chooses between them
- If none of the conditions are met, it outputs nothing or the phrase *unhandled_ifs*

Advanced Example 4.4 – Variables within Ifs and Learns

So far, we've used variables in the left hand side of *Ifs* and *Learns*, before the = sign, such as *friendly=yes* and *formality<=2*. You can also use variables on the right side of the =, <, >, etc signs, but you have to surround them with $ signs, as you did for *outputs*. If not, the bot will not know it's a variable. Here is a simple example:

Type	Label	D.	Text	If	Learn	Goto	Acc
output start	welcome		Hi there, I'm your fourth conditional bot!				
input	friendly		((hello))		history="$history$ nice"	reply	60
			((hello)) there				
input	unfriendly		((go away))		history="$history$ mean"	reply	60
output	reply		You have been $history$. Yay!				
			You've been $history$. Boo!	mean_age < $nice_age$			
phrase	hello		hello / hi / hey		nice=yes		
phrase	go away		scram / go away		mean=yes		

In this example the *history* variable keeps a running list of whether you have said something nice ("hello", "hi" or "hey") or something mean ("scram" or "go away"). Each time you say one of these, the *history* variable learns itself with the word "nice" or "mean" tagged on the end.

There's also an example using variable ages in the *If*. The comparison *mean_age<$nice_age$* will be true if you were mean more recently than you were nice, and the *output* will have "Boo!" at the end, otherwise "Yay!" There are better ways of doing this, but it's just for an example.

It may seem confusing that the variables on the left side of the equation do not have $, but the variables on the right side do. In fact, you could put them on the left side as well, such as *$nice$=yes* or *$mean_age$<$nice_age$*. We avoided this syntax up till now in order to keep things as simple as possible.

The *If* and *Learn* column may look a bit like a programming language, and they are in a basic way. They support = and comparisons and *and* and *or*. They are more flexible than normal programming as they don't always require quote marks or variable signifiers like $ signs. But they are also much more limited, as they don't support mathematical operations like $*nice_age$+2*.

Important points:

- the *If* and *Learn* columns can variables on the right side of the =, <, > etc sign as well
- the $ signs are optional on the left side of the =, <, > etc
- the $ signs are required on the right side of the =, <, >, etc
- comparisons like *mean_age<$nice_age$* are supported
- variables within strings like *"$history$ nice"* are supported
- no other operations are allowed

Section 5 – Structure

All of the bots in this manual are very simple in their structure. All but one have all their *inputs* under the *output start*. The only exception was the bot who asked about peas. That bot showed how a more complex branching structure could be developed.

That branching structure is fairly easy to implement and understand for simple bots, but can get unwieldy when there are dozens or hundreds of *inputs* and *outputs*. This section shows alternative ways to structure bots using *Ifs* and variables.

At the end of the section is a summary of how all the different columns fit together, and a discussion of when different approaches are appropriate.

Example 5.1 – Multiple Gotos

So far, all our *input* lines have no *Learn* and just a single entry in *Goto* columns, or no *Goto* at all (then the bot goes to the *output* with the same label). But an *input* can have more than one *Goto*, each with an *If* and a multiple *Learn*. The shading shows how the *If* column is tied to the *Learn* and *Goto* for *inputs*. Like this:

Type	Label	Desc	Text	If	Learn	Goto	Acc
output start	welcome		Hi there, I'm your first structural bot!				
input	greeting		((hello))	friendly=yes	mood=good	reply_good	60
			((hello)) there	friendly=no	mood=bad	reply_bad	
			((hello)) to you				
			((go away))				
phrase	hello		hi / hello / hey		friendly=yes		
phrase	go away		go / go away		friendly=no		
output	reply_good		That's $mood$.				
output	reply_bad		I feel $mood$.				

For *inputs*, both the *If* and *Learn* relate to the *Goto*. In the examples so far, we have had just one *Learn* and one *Goto*. This is just an extension of that, with added *If* conditions.

If you say “hello” or “hey there” or “hi to you” then the variable *friendly* will learn the value *yes*. Then the bot will look at all the *Gotos* for the *input* labelled *greeting* and process them in order. The first *Goto* has the *If* condition *friendly=yes*. In this case, it is true, *friendly* does equal *yes*, so the variable *mood* will learn the value *good* and the bot will go to the *output* labelled *reply_good*.

Important points:

- *inputs* can have more than one *Goto*
- each *Goto* can have an *If* and a *Learn*
- the *Gotos* are processed in order, the first matching one is used
- for *inputs*, the *If* and *Learn* columns relate to the *Goto*, this is different from *phrases* and *outputs* where the *If* relates to the *Text* and there can only be one entry in the *Learn* column
- an *input* can have more lines with *Gotos* than *Text*
- the separator / is also supported for all three columns *If*, *Learn* and *Goto*

Example 5.2 – Dynamic and Unhandled Gotos – Informational Bot

You can also use variables within *Gotos*. We call these dynamic Gotos. They are a powerful technique enabling you to do a lot of the recognition in *phrases* rather than *inputs*. This example is the basis of an informational bot, one that can answer questions on a website or in a game.

Rather than having lots of different *inputs* for each question ("what is a computer?", "tell me about Cleverscript", etc), it has one different *input* for each query type (in this case, "what is..." type enquiries). The ((topic_list)) phrase recognises allowed topics and sets the *$topic$* variable, which then tells the bot to go to the appropriate *output*, formed by they dynamic *Goto topic_$topic$*.

Type	Label	D	Text	If	Learn	Goto	Acc
output start	welcome		Hi, I'm your second structural bot! Ask me a question.				
input	what_is_something		((what is)) ((topic_list))			topic_$topic$	40
output	topic_computer		A computing device.				
output	topic_cleverscript		Cleverscript made me!				
output	topic_name		My name is ((bot_name)).				
output	unhandled_goto		Something went wrong.				
phrase	what is		what is / what's / whats				
			tell me about / tell me				
phrase	topic_list		((computer))->computer		topic		
			cleverscrupt				
			your name->name				
phrase	computer		computer / laptop / machine				
phrase	bot_name		Evie				

This bot can only answer a limited set of questions such as "what is your name?" But it can be easily extended. For every new query topic, you just need to add one line to the *phrase* labelled *topic_list*, and one *output* to give the answer. And to get your bot to answer a whole new type of question such as "what is your favourite ...", you need to add a new *input*.

The danger with this approach is that you may mistype a topic, and the bot will end up trying to go to a *Goto* which doesn't exist. This sort of error would not be flagged up when importing. Normally, if this happened, it would show the user an error message like "Error: Could not find the goto 'topic_comppuuter'". But you can hide the error message by providing an *output* labelled *unhandled_goto* as above. The actual error is still available in the *$errorline$* variable. (The *unhandled_goto output* will also handle blank inputs if *blank_input output* is not provided.)

To see this in action, ask "what is cleverscript?". It will match the mistyped word "cleverscr**u**pt", learn "cleverscrupt" as *topic* and try to go to an *output* labelled *topic_cleverscr**u**pt*. This doesn't exist so instead you'll see the *unhandled_goto*.

Important points:

- dynamic *Gotos* containing variables are a powerful way to set up informational bots
- if there are no *Gotos*, or if none of the *Goto If* conditions are met, then the bot goes to the *output* with the same label, or else goes to the output *unhandled_goto* or else gives an error

Example 5.3 – Dynamic Phrase Labels

From March 2013 you can also use variables within *phrase* labels. These are known as dynamic *phrase* labels. It allows for the bot above to be more streamlined. In the bot above, for every new question, you have to add a choice to *topic_list* and a new *output*. This works well, until you want to modify all the *outputs*, to add a debugging message for example. Dynamic *phrase* labels gets around this by having just one *output* which picks a *phrase* based on a variable. Like this:

Type	Label	D	Text	If	Learn	Goto	Acc
output start	welcome		Hi, I'm your third structural bot! Ask me a question.				
input	what_is_something		((what is)) ((topic_list))				40
output	what_is_something		((topic_$topic$)) ((debug))				
phrase	topic_computer		A computing device.				
phrase	topic_cleverscript		Cleverscript made me!				
phrase	topic_name		My name is ((bot_name)).				
phrase	what is		what is / what's / whats				
			tell me about / tell me				
phrase	topic_list		((computer))->computer		topic		
			cleverscrupt				
			your name->name				
phrase	computer		computer / laptop / machine				
phrase	bot_name		Evie				
phrase	debug		Topic=$topic$.				
phrase	unhandled_phrase		I don't know.				

This bot is very similar to the one above except the answer comes from a dynamically referenced *phrase* instead of a dynamic *Goto*. And as above, if you ask “what is cleverscript?” it will try to output a non-existent *phrase* labelled *topic_cleverscrupt*. So as with the *unhandled_goto* above, you can also define a *phrase* labelled *unhandled_phrase*. This will be delivered instead. Otherwise your bot will deliver the missing *phrase* label directly in its reply.

Another way to handle this is by adding an *If* condition to your *output*, which says *PhraseExists:$topic$* or *PhraseMissing:$topic$*. These evaluate to true or false.

Dynamic *phrase* labels can also be used for matching the user input. However they are inefficient compared to normal *phrases* as they involve a database lookup. This is okay if it happens a few times for an output, but could slow things down if had hundreds of different dynamic *phrase* labels being checked for every user input.

Important points:

- dynamic *phrase* labels containing variables provide another powerful technique
- dynamic *phrase* labels allow you to have a single *output*
- if a *phrase* label does not exist, the *phrase* labelled *unhandled_phrase* is delivered instead
- or else the bot will output the missing *phrase* label directly in its reply
- dynamic *phrase* labels can also be used for input but are not as efficient as normal *phrases*

Example 5.4 – Conditional Inputs – Game Bot

This example shows another way to implement a branching structure. This puts all the *inputs* under a single *output start*, but has a special **If* condition attached to the inputs. This special **If* determines if the bot should even consider that *input* at all.

The following example demonstrates the special **If*, along with multiple *Gotos*, optional and unimportant phrases, and wildcards.

Type	Label	D.	Text	If	Learn	Goto	Acc
output start	welcome		Hi there, I'm your third structural bot! You are in a maze. Go to the correct room and shout.		room=1		
input	go_straight		((!?go)) ((straight))	room=1	room=2	good_way	60
				room=3	room=4	good_way	
						bad_way	
input	go_left		((!?go)) left	room=2	room=3	good_way	60
						bad_way	
input	go_other		((go)) ~			bad_way	40
input	shout		((shout))	*room=4			40
						done_maze	
output	good_way		Good choice. You're now in room $room$, keep going!				
output	bad_way		Ooops. Not quite.				
output	done_maze		Well done! You won!				
phrase	straight		straight / up / forward				
phrase	go		go / turn / move / head				
phrase	shout		shout / yell / scream				

This bot implements a very simple game. There are three rooms, and you start in room 1, with the variable *room=1*. There are four *inputs*. The *inputs* starting with *go_* are always active. The first two, *go_straight* and *go_left* have multiple *Gotos*. They only allow you to progress if you say the correct direction for the room you are in.

The *input* labelled *shout* is only active when you are in the fourth and final room. Otherwise, the bot completely ignores it. If you say "shout" in rooms 1, 2 or 3, it will be considered an invalid input, and you'll go back to the beginning of the game. This is achieved with a special *If*. The * in front of the *If* tells the bot to only even consider the *input* if the *If* is true.

To win the game, say "go straight" (*room=2*), then "go left" (*room=3*) , "go straight" again (*room=4*) and then "shout". Any wrong direction at any stage will give you the *output* labelled *bad_way*. Anything completely unexpected, like "hello", or "shout" in the wrong room will send you back to the *output start* and restart the game.

Note that there is no *Learn* or *Goto* tied with this special **If*. It stands alone. The shading shows visually how the special **If* is tied to the *input* and not the *Goto*.

This game without variables
This game could have been programmed quite easily without variables, by using the branching structure as in the pea-obsessed bot, having a different *output* for each room. However, you would have had to duplicate the *input* labelled *go_straight* under both the *room1* and *room3 outputs*, which is okay for this game, but less practical if you have 100 rooms or game states.

Without variables, an *input* has to be always active (under the *output start*) or else only active under one specific *output*. It's not possible to have an *input* active for only some of the *outputs*.

What multiple Gotos give you
Multiple *Gotos* essentially allow you to do this, once you've introduced some variables. With mutiple *Gotos*, an *input* can be active in two or more, but not all *outputs*.

What special *_If_s give you
The special **Ifs* take this even further. The *input* labelled *shout* above does not even need multiple *Gotos*. It is simply turned off when you are not in room 4. This means that you don't have to consider how this particular *input* should behave when the game player is in one of the other rooms.

This is mildly useful in this example, but very useful if you have 100 rooms or game states. It means you can add new *inputs* specific to a handful of game states, and ignored for the rest. You don't have to consider the implications of overriding some other *input*. Not having to add lots of *Ifs* for multiple *Gotos* also reduces the risk of a dynamic *Goto* error.

Important points:
- *inputs* can have a special **If* preceded by a * which determines if they are considered at all
- the special **If* has no *Learn* or *Goto*
- normal *If/Learn/Gotos* must be placed on subsequent lines of the spreadsheet

Example 5.5 – Output Borrowing

The *Goto* column is also be used with *outputs*. It allows one *output* to borrow the *inputs* of another *output*. It essentially allows an *output* to *Goto* another *output* to get more *inputs*. Feel free to take a couple minutes to digest that one.

Output borrowing is a more direct way for an *input* to be active in two but not all *outputs*. This example shows another version of the game above, without any varibles and with only one *If*.

Type	Label	D.	Text	If	L.	Goto	Acc
output start	welcome		Hi there, I'm your fourth structural maze bot! Type 'start' and find your way.				
input	go_other		((go)) ~			bad_way	40
input	start		start			room1	40
output	room1		Welcome to room 1.				
input	go_straight		((!?go)) ((straight))	output_label=room3		room4	60
						room2	
output	room2		Welcome to room 2.				
input	go_left		((!?go)) left			room3	60
output	room3		Welcome to room 3.			room1	
output	room4		Welcome to room 4.				
input	shout		((shout))			done_maze	40
output	bad_way		Ooops. Back to room 1.			room1	
output	done_maze		Well done! You won!				
phrase	straight		straight / up / forward				
phrase	go		go / turn / move / head				
phrase	shout		shout / yell / scream				

This version has a separate *outout* for each room. Notice that the *output* labelled *room3* borrows from *room1* in the shaded *Goto* column. This means that the *input* labelled *go_straight* is active in *room1* and *room3*. Consequently it needs an *If* to know whether it should go to *room2* or *room4*. The *output* labelled *bad_way* also borrows from *room1*. Whenver you give a bad direction, you'll return to *room1*. Otherwise *bad_way* would have need lots of *Ifs* as well.

In this verison, you have to type “start” to get into *room1* in the first place. This is because if *go_straight* was under the *output start* it would be active in all rooms. The *outputs bad_way* and *room3* could have also directly borrowed the *input* labelled *go_straight*. In this example game, it would have had the same effect.

Important points:

- *outputs* can borrow *inputs* from other *outputs*
- to do this, put the *output* labels in the *Goto* column
- an *output* can borrow as much as it would like, put each on a separate line or separated by /
- you can also borrow directly from other *inputs*
- borrowings are not related to the *If* and *Learn*

Example 5.6 – Conditional Outputs

In a similar way to conditional *inputs*, you can also put a special **If* in an *output*. The *output's* special **If* is checked just after the *input's If/Learn/Goto*. If the special **If* is false, then that *Goto* is skipped.

Type	Label	D.	Text	If	L	Goto	Acc
output start	welcome		Hi there, I'm your fifth structural maze bot! Type 'start' and find your way.				
input	go_other		((go)) ~			bad_way	40
input	start		start			room1	40
output	room1		Welcome to room 1.				
input	go_straight		((!?go)) ((straight))			room4	60
						room2	
output	room2		Welcome to room 2.				
input	go_left		((!?go)) left			room3	60
output	room3		Welcome to room 3.			room1	
output	room4		Welcome to room 4.	*output_label=room3			
input	shout		((shout))			done_maze	40
output	bad_way		Ooops. Back to room 1.			room1	
output	done_maze		Well done! You won!				
phrase	straight		straight / up / forward				
phrase	go		go / turn / move / head				
phrase	shout		shout / yell / scream				

This bot is exactly the same as the one before. You would use this method when you have lots of *inputs* going to the same *output* only if some *If* condition is met. With this method, you can avoid copying and pasting the *If* condition next to the *inputs*. Instead you can have it just once as a special **If* associated with the *output.*

Note that each variation in an *output* can also have its own *If* and *Learn*. So you have to put the special **If* at the end of all the other *Ifs*, or next to any variation which doesn't already have an *If*.

Using this feature makes it more likely to cause an error where none of an *input's Gotos* will be viable. If this happens, then the bot will instead to go the *output* labelled *unhandled_goto*, or will give an error.

Important points:

- *outputs* can have a special *If* preceded by a * which determines if they are used at all
- the special *output If* has no *Learn*
- the special *output If* can be put on any line that doesn't already have an *If*

Example 5.7 - If and Learning Summary

There are now several different ways the *If* and *Learn* columns are used, depending on whether they appear in an *output*, *input* or *phrase*. This colourful example shows how it all fits together.

1) After you say something to the bot, it goes through each of the last *output's inputs* (and any borrowed ones, shown in green) looking for a match. The first thing it checks is the *input's* special **If* (dark blue below).
2) For each viable *input*, the bot tries to parse what you said into *phrases*. It finds the best matching *input* and starts processing it, first looking at the *input's phrases*.
3) For *phrases*, it first checks the *If* conditions and skips any variations which are false. The *Learn* column contains a single variable name with an optional default value. The actual value learned comes from the *Text* column, either the *Text* itself or the value after the ->. This kind of learning happens first while processing the input (shown in reddish, using the short-cut method where a single *Learn* appears with the first line of *Text* only).
4) Immediately *after* learning any *phrase* variables, the bot checks through the *input's Ifs*. If an *If* is true, then the *Learn* is done and the *Goto* is followed (shown in aqua). The *Learns* are executed before the *Goto*, so the *Goto* can dynamically reference learned variables. The bot follows the *Goto* and immediately checks if the *output* has a special **If* (also shown in aqua below). If that is true, the bot uses the *output*. If it's false, the bot unwinds, unlearns any *Learn*, and tries the next *Goto*.
5) If none of the *input's Gotos* work, or if the *Goto* is missing or goes to a non-existent *output*, then the bot looks for an *output* labelled *unhandled_goto* instead, or else outputs an error message.
6) The bot has reached the *output*. For *outputs*, the *If* and *Learn* columns are tied to the *Text* column. Only if the *If* evaluates as true, is the *Text* output and the variables learned (shown in yellow). This learning happens *after* the *Goto* above.
7) The *output* can reference *phrases* and *output phrases*. These *phrases* can also have *Ifs* tied to their *Text* and a single *Learn* (shown in brown).

Type	Label	D.	Text	If	Learn	Goto	Acc
output start	welcome		Hi there!				
input	greeting		((hello))	*mood#awful			60
			((hello)) there	friendly=yes or friendly=maybe	mood=good	bot_mood	
			((hello)) to you	friendly=no	mood=bad	bot_mood	
			((go away))				
output	bot_mood		That's ((nice)).	mood=good	nicecount=+1	welcome	
			That's mean.	mood=bad	meancount=+1		
				*friendly#very			
phrase	hello		hi->yes		friendly		
			hello->maybe	nicecount<10			
output phrase	nice		nice	nicecount<2	imnice=yes		
			so nice	nicecount>=2	imsonice=yes		
phrase	go away		go / go away		friendly=no		

The special **Ifs* and *output* borrowing shown here don't do anything. They are for example only.

Structural Approaches

As you've seen in this section, there are several ways to implement the same bot. The different methods are generally trade-offs between readability and complexity. An experienced programmer may organise a bot using lots of *Ifs* and *Learns*. A novice may like to diagram their bot first, and then implement each *output* and *input* separately. Here are some possible approaches:

Branching structure:
This structure was introduced in one of the first examples, for the bot who asked about peas. It is recommended if you are non-programmer who is getting used to Cleverscript, of if your bot is very sequential and specific, with each *output* expecting different *inputs*. This method does not require any *Ifs* or *Learns* but can still produce complex bots. The disadvantage is that you may find yourself copying and pasting a lot as the bot grows.

Output borrowing:
Output borrowing allows for less copying and pasting, while still keeping a branching structure. It is also useful if you are creating several bots and would like to have a library of *outputs* and *inputs* which they can all use. You can put all your common *outputs* and *inputs* into one spreadsheet, and your bot-specific *outputs* and *inputs* into separate spreadsheets, borrowing as needed.

Multiple gotos and special Ifs:
If you are comfortable with variables, then putting all your *inputs* under a single *output start* can avoid a lot of copying and pasting, and help to maintain consistency. You will need at least one topic or state variable to remember what the user is doing. Though try to keep the number of regularly used variables to a minimum, as a complex bot with many variables can be difficult to debug.

The special **Ifs* for *inputs* and *outputs* can streamline bots with lots of multiple *Gotos*. If your variables have readable names and values (such as *state=room1* or *topic=insurance*) then your spreadsheet will be easy to follow.

You can also colour code your spreadsheet, as in the example below. You only need to export it to a tab-delimited file when creating a bot. Your working copy can have all the features of your spreadsheet software.

Single input or output:
At the other end of the spectrum, it is possible to develop a bot with just a single *input* and a single *output*. For example, you could take an existing bot with 10 *inputs*, and put all those *inputs* into a single *phrase* labelled ((inputtype)) which learns a variable called *$inputtype$*. Your single *input* would just include this *phrase* ((inputtype)). Then you could combine all your *outputs* into a single *output* and use *If* conditions (such as *inputtype=friendly*) to give the correct response.

Or as an alternative to *If* conditions, you could use a single *output* which references a dynamic *phrase* label such as ((answer_$inputtype$)), and then each answer can be a separate *phrase*.

Doing a bot this way would produce an organised bot, but would lose out on a lot of the functionality built into Cleverscript like *Gotos*. It is described here just as an example. The method you choose will probably end up as a mixture of all the ones above.

Section 6 – Clever Data for Small Talk

Existor's first and foremost creation is www.cleverbot.com. Cleverbot is a very popular website and Smartphone application. Unlike Cleverscript, where all *outputs* are planned and scripted, Cleverbot learns from the people who talk to it and responds with what it considers best. Consequently, it is quirky, unpredictable and very entertaining.

Cleverbot began in the late 1980s and currently has a huge database of over 120 million lines of conversation. It has won the Loebner Prize twice and does well on Turing Tests.

In February 2013 we added a miniature version of Cleverbot to Cleverscript. This means that, along with all its scripted responses, Cleverscript can now also do some general low-level small talk! Things like “How are you?”, “I'm fine thanks”, “That's great”. And you don't need to write a script for any of it.

We call this new feature “Clever Data”. It uses a stripped down version of the Cleverbot engine, along with a much smaller data set. Instead of 120 million lines of potential conversation, Clever Data uses just a few thousand lines. This section describes how to use it.

Example 6.1 – Clever Data Fallback

On the Upload page, you can now choose to include some Clever Data in your bot. We will eventually provide a drop down with several different sizes and themes of data. For now, you can choose a small set of data from Cleverbot.

To see how it works, try uploading the example below, and be sure to include some Clever Data. You can try chatting to them on the Cleverscript website once you have registered.

Type	Label	Description	Text	If	Learn	Goto	Accuracy
output start	welcome	The first thing the bot will say.	Hi there, I'm your first small talk bot!				
input	hello	User says hello	hello				75
output	hello	Bot replies hello.	Hello to you too.				

This example is like example 1.2 but without the fallback *input* (the one previously labelled *anything*). The bot opens with “Hi there, I'm your first really chatty bot”. If you say “hello” it will say “Hello to you”.

If you say anything else, it would normally go back to its *output start* and give its opening line again. But with the Clever Data included, it will instead give a conversational output. If you say “hi there”, it will reply something like “How are you?”

Note that when testing this bot on the Cleverscript website, you are only chatting to about 10,000 lines of Cleverbot data. When you publish your bot via the API, it will use more like 1 million lines and so will **it seem much more intelligent**.

Important points:

- include Clever Data in your bot to let it do small talk
- if none of the *inputs* match, then the bot will reply from its Clever Data
- the Clever Data in www.cleverscript.com is much smaller and less intelligent than when you publish your bot and chat it to though the API

Example 6.2 – Clever Data Variables

You can get more control over the Clever Data replies using two new internal variables: *$clever_accuracy$* and *$clever_output$*.

Type	Label	D.	Text	If	Learn	Goto	Acc
output start	welcome		Hi there, I'm your second small talk bot!				
input	hello		hello				75
input	anything		anything				0
output	hello		Hello to you too.				
output	anything		$clever_output$	clever_accuracy>30			
			Sorry, I don't have a response for that. I only scored $clever_accuracy$%.				

This example is like the one above but with the fallback *input* put back in. When you say "hello" it will still reply "Hello to you too."

If you say anything else, it will trigger the *input* labelled *anything* (which has 0 *accuracy*) and go to the *output* labelled *anything*. This *output* has two possibilities. If the *clever_accuracy* variable is more than 30%, the bot will deliver the *clever_output,* or else if will say "Sorry, I don't have a response for that. I only scored 23%."

The Cleverbot engine inside your bot compares your whole conversation so far to previous conversations in its database. It outputs a line from the best-matching conversation. The *clever_accuracy* variable measures how well it actually matched. An accuracy of 100% would mean that your conversation exactly matches one in the Clever Data database. In practice, this is very unlikely to happen. In fact, the accuracy scores from Clever Data are generally lower than accuracy scores for Cleverscript because they are comparing the whole conversation, not just the last thing you typed. Even an accuracy of 30% can produce a reasonable reply, and the average accuracy will increase for bigger Clever Data databases.

The *clever_output* variable contains the actual output from Clever Data. Unlike the example above, you now need to explicitly output this.

Important points:

- the variable *clever_output* contains the Clever Data reply
- the variable *clever_accuracy* stores how well the Clever Data reply matched
- *clever_accuracy* scores are generally low

Example 6.3 – Reaction and Emotion

The avatar at www.existor.com also uses the full 140 million rows of Cleverbot data. Notice that she smiles or frowns when she says things. This is because every Cleverbot response comes with *emotion* and *reaction* attributes. Clever Data has inherited this useful feature.

It means that nearly every Clever Data reply also includes a pre-computed *emotion* and *reaction*. For example, if the Clever Data reply is "who are you?" the *emotion* might be *curious*. These are available to your script as variables. Here's an example:

Type	Label	D.	Text	If	Learn	Goto	Acc
output start	welcome		Hi there, I'm your third small talk bot!				
input	hello		hello				75
input	anything		anything				0
output	hello		Hello to you too.				
output	anything		$clever_output$ Reaction: $reaction$ with tone $reaction_tone$. Emotion: $emotion$ with tone $emotion_tone$.				

When you say "hello" to this bot, it will reply with the *output* labelled *hello* and say "Hello to you too." If you say anything else like "hi there" then the *output* labelled *anything* will deliver the Clever Data reply via the variable *clever_output*. Clever Data will also set six internal variables: *reaction*, *reaction_tone*, *reaction_degree*, *emotion, emotion_tone* and *emotion_degree*.

The *reaction* and *emotion* are text strings, things like *winking* or *happy*. A full table of all the possibilities is shown below. Each one also has an accompanying tone, which is a number from -2 (for a very negative *emotion* or *reaction*) to 2 (very positive).

These variables are intended to be used for controlling visual avatars, because when you are ready to put your bot on your own website, you may want to give it a face as well. This could be a single image, or a set of images which change depending on the *emotion* or *reaction*.

There are approximately 50 *reactions* and 70 *emotions*, so in theory there are over 3000 unique combinations of *reaction* and *emotion*, and your avatar could have a different look for each (though in practice many of the combinations would not occur). Furthermore *reaction_degree* and *emotion_degree* give a rough estimate of the degree or strength of the reaction/emotion. Alternatively there are 25 combinations of *emotion_tone* and *reaction_tone*. (In fact, this is how the face at www.existor.com is controlled. It uses sophisticated custom software to morph between different *emotion* and *reaction* images.)

These six variables are only set when your bot is asked to deliver a Clever Data output or compute the Clever Data score. This means that you can also use the variables in your scripts. So that when your bot gives a scripted reply, you can set an *emotion* or *reaction* attached for controlling an avatar.

Important points:

- Clever Data includes a *reaction* and *emotion*, listed above
- it also provides *reaction_tone* and *emotion_tone*, numbers from -2 to 2
- it also provices *reaction_degree* and *emotion_degree* as rough percentages from 0 to 100
- these variables are only set when Clever Data is used in an interaction
- this means that you can also set them in scripting and they won't be overwritten

Reactions and emotions

Below is a list of all the reactions and emotions that can occur within Clever Data along with their tones. Please note that these reactions and emotions are mostly programmatically computed, so they may not reflect who a human would actually feel if having the same conversation.

Reactions	Tone	Emotions	Tone	Continued	
aah	0	agreeable	0	singing	0
agreement	0	alert	0	sleepy	0
amazed	2	amused	1	smug	0
annoyed	-1	angry	-1	stubborn	0
appreciation	1	apologetic	-1	supportive	1
belief	0	argumentative	-1	sure	0
confused	-1	assertive	0	sweetness	1
cool	0	bored	-1	sympathy	1
crying	-1	calm	0	thoughtful	0
disagreement	-1	concerned	0	tired	0
disappointed	-1	contemplative	0	tongue out	-1
disbelief	0	curious	0	uncomfortable	-1
disgust	-2	dancing	0	unsure	0
disinterested	0	determined	0	very happy	2
displeased	-1	devious	-1	very sad	-2
eek	-2	didactic	0	victorious	0
embarrassed	-1	distracted	0	winking	0
forceful	0	doubting	-1	worried	-1
frowning	-1	excited	2		
frustrated	-1	flirty	0		
genuine smile	1	forgetful	0		
giggling	1	furious	-2		
ha	1	gentle	0		
impressed	1	grumpy	-1		
indignation	-1	guilty	0		
infuriated	-2	happy	2		
interested	1	hatred	-2		
knowing	0	joking	0		
look down	0	jumpy	0		
look left	0	lazy	0		
look right	0	love	2		
look up	0	mean	-1		
nasty goodbye	-1	mocking	-1		
nasty laugh	-1	modest	0		
nice goodbye	1	naughty	-1		
nice hello	1	negative	-1		
nice laugh	1	nice	1		
none	0	none	0		
pleased	1	nosey	0		
relieved	0	positive	1		
reluctant hello	0	proud	1		
sarcastic smile	-1	questioning	0		
scared	-1	relaxed	0		
shocked	-2	reluctant	0		
sigh	0	righteous	0		
sneering	-1	robotic	0		
sniggering	0	rude	-1		
surprised	0	sad	-1		
uncomfortable	-1	sarcastic	-1		
unimpressed	0	serious	0		
uninterested	0	shouting	-1		
upset	-1	shy	0		
wry smile	0	silly	0		

Advanced Example 6.4 – Influencing Clever Data

There are a few other variables relating to Clever Data. The variables *interaction_1* up to *interaction_20* store the last twenty user/bot interactions. The variable *interaction_1* stores the most recent thing the user said, and *interaction_1_other* stores what the bot replied. These variables provide Clever Data with its conversation history, so it can match the current conversation against previous ones.

The variable *clever_match* allows you to directly influence how the Clever Data chooses its best match. You can use the variable *clever_match* to tell the Cleverbot engine to favour a reply containing your chosen word or words.

Type	Label	D.	Text	If	Learn	Goto	Acc
output start	welcome		Hi there, I'm your fourth small talk bot!		clever_match=human		
input	hello		hello				75
output	hello		Hello to you too.				

This bot is like the first example in this section, except that it will favour Clever Data replies that contain the word “human”. It's really just a small nudge in the right direction.

For instance, if you ask this example bot, “are you a computer?”, it has several possible replies including “I'm not sure.”, “Of course.” and “No, I'm a human.” Normally, the Cleverbot engine is roughly equally likely to choose any of these responses. But if the *clever_match* variable is set to “human”, then it will favour the latter one.

You can therefore use the *clever_match* variable to give the Cleverbot engine at hint at the current topic of conversation. Then if the user asks something not covered by your script, the Clever Data output will attempt to stay on topic.

Later on in the conversation the effect becomes less noticeable because there are other factors determining the best reply, such as the context provided by the conversation history.

This effect is more noticeable with commonly asked questions in bigger data sets, because the short-list will be more uniform and the presence of the *clever_match* word or phrase will have more of an impact.

Important points:

- the variable *clever_match* can influence the Clever Data output
- you can use this to hint at a topic for the small talk which Clever Data engages in
- the effect is more noticeable for common questions in big data sets early in a conversation

Advanced Example 6.5 – Influencing Clever Data Even More

From July 2013 there is a further way to influence Clever Data, using a new *Type* called *clever adjust*. As above, this adjustment happens in the latest stages of Clever Data, when your bot is trying to choose between a handful (20-50) of candidate rows.

Clever adjusts are tied to a specific *input*. However they behave more like a *phrase* as they learn a percentage adjustment to the *clever_accuracy*. For example:

Type	Label	D.	Text	If	Learn	Goto	Acc
output start	welcome		Hi there, I'm your fifth small talk bot!				
input	hello		hello / how are you			converse	75
clever adjust	hello_adjust		((good))->-5				60
			how are you->10				
input	fallback		anything else			converse	0
output	converse		$clever_output$				
phrase	good		good / ((really)) good / great / fine				
phrase	really		really / very				

If you type “hello” or “how are you”, this will trigger the *input* labelled *hello*. As usual, this will follow the *Goto* and the bot will start to deliver the *output* labelled *converse* which tells it to deliver the Clever Data variable *$clever_output$*. At this stage the bot will go back to the *input* and apply the first (and only the first) *clever adjust* it finds, which is *hello_adjust* in this case.

If the potential reply matches with 60% accuracy or more, then an adjustment is applied. If it matched against *((good))*, then the score is reduced by 5%. So if the *clever_accuracy* would have been 48%, it will become 43%. Similarly if it matches “how are you”, it will be increased by 10%.

It means that this bot is likely to reply to your greeting with something like “how are you?” and very unlikely to say “good”, “very good”, “really good”, “great” or “fine”. Note that after the first time it says “how are you?” there is a punishment for repeating the same thing again, so it may well not say “how are you?” again but “how are you today?” or similar.

With a negative percentage, this feature allows you to essentially rule out certain outputs and have the Clever Data automatically choose the next best alternative, without you having to script the alternatives, or filter the output yourself (filtering is discussed in the next section). With a positive percentage, you can favour chosen outputs.

Important points:

- the new *Type* called *clever adjust* can be used to adjust scores at the final stage
- you can use it favour or demote certain replies
- the *clever_adjust* does not need a *Learn* as for efficiency, no actual learning is done
- if the variable *$clever_output$* is included within other text, this feature may not work

Section 7 – Filtering

Filtering is a new Cleverscript feature introduced in February 2013. It allows you to modify the user's input before it is matched against your *inputs*, and to modify the bot's output after it has been formed.

For example, you could use an *input filter* to strip extra politeness from whatever the user says, so that your individual *inputs* don't have to worry about it. And you could use an *output filter* to change the output from Clever Data, replacing the word “human” with “alien” or anything else.

As with all *phrases*, *filters* operate on parts of words. So a new syntax involving a colon : is introduced to make sure that *filters* only match whole words.

Example 7.1 – Input Filtering to Remove Text

An *input filter* is really just a special type of *input*. It can be associated with the *output start* or with any other *output*. It has *Text* and an *Accuracy* but no *Goto*. In its simplest form, it uses the *unimportant phrases* feature (the ! inside a phrase reference) to remove text from the user's input. Here is an example:

Type	Label	Desc	Text	If	Learn	Goto	Accuracy
output start	welcome		Hi there, I'm your first filtering bot!				
input filter	please filter		((:!please:))				75
input	say hello		say hello				75
output	say hello		Hello.				
phrase	please		please / plz				

The *input filter* in this example, will strip the string “please” or “plz” from the user's input, before matching it against the *input* labelled *hello*. So whether you type “say hello” or “say hello please” or “plz say hello”, the *output* will be the same. The bot will reply “Hello.” with 100% accuracy.

The *Accuracy* determines how well the entire *input filter* must match the user's input before being activated. There are implied wildcards around the filter, so actually it is comparing *~ ((!please)) ~* against the user's input. If that matches at least 75%, then it removes the unimportant *((!please))* phrase from the user's input. (Internally, it is not really adding the wildcards; it has a more efficient method, but is behaving in a similar way.)

Each filter runs multiple times, so even if you said “please say hello plz please please!”, all the “pleases” would be removed. It is therefore recommended not to use too low an *Accuracy* for *input filters*, so that they don't mistakenly remove something important. The 75% here means that some very close mis-spellings like “pleasse” would also be removed.

Note that *input filters* would normally also remove parts of words, so if you typed “say helplzlo”, it would still remove the “plz”. This is sometimes useful, but can often lead to unintended confusion. So the : marks tell the bot to only remove/replace a whole word. The : at the beginning matches the start of a word (or the start of the input) and the : at the end matches the end of a word (or the end of the input). In this case, the “plz” in “say helplzlo” will not be removed.

Input filters and *phrases* within *input filters* can also have *Ifs* and *Learns*. These behave as they do in *phrases*.

Important points:

- *input filters* can be used to remove unnecessary text from the user's input
- *input filters* can remove parts of words
- use : at the beginning and/or end to remove/replace whole words only
- any unimportant phrases such as *((!please))* are removed
- *input filters* run multiple times
- *input filters* do not have a *Goto*
- *input filters* can have *Ifs* and *Learns*, which behave in the same way as *phrases*

Example 7.2 – Filters Between Inputs

You can have more than one *input filter* and they can appear before or after *inputs*. *Input filters* are processed in the order they appear in the spreadsheet, changing the user's input as they go. They can also have *special Ifs* attached to them, making them conditional, as shown in the rather contrived example below.

Type	Label	Desc	Text	If	Learn	Goto	Accuracy
output start	welcome		Hi there, I'm your second filtering bot!		mean=no		
input	unfriendly		scram / go away		mean=yes		75
input filter	please filter		((:!please:))	*mean=no			75
input	hello		say hello				75
output	unfriendly		That's not nice.				
output	hello		Hello.				
phrase	please		please / plz				

The *input filter* in this bot appears after the first *input* labelled *unfriendly*. So if you say “go away please” it won't strip off the “please” and won't match the *input*. But if you say “say hello please”, then the “please” will be removed before the bot gets to the *input* labelled *hello*, and the bot will reply “Hello.”

If you say “go away”, it will learn the variable *mean* as *yes* and the *input filter* will become inactive because of its *special If* condition, which checks if the variable *mean* is *no*. So if you then say “say hello please”, the “please” won't be removed, the *input* labelled *hello* won't match and the bot will return to its *output start*. This will set *mean* back to *no* and the whole cycle can restart.

During processing, the variable *filtered_input* contains the filtered input at that line. At the end of processing, *filtered_input* contains the input with all filters applied.

Important points:

- *input filters* are applied in the order they appear in the spreadsheet
- *input filters* have no goto but can have special Ifs
- the variable *filtered_input* contains the filtered input while processing

Example 7.3 – Input Filtering to Replace Text

Input filters don't just have to remove text. They can also replace it with something else. This is accomplished with phrase learning. If the unimportant phrase within the *input filter* has a *Learn*, then the learned value is used as the replacement text.

Type	Label	Desc	Text	If	Learn	Goto	Accuracy
output start	welcome		Hi there, I'm your third filtering bot!				
input filter	please filter		((:!please:))				80
input filter	say filter		((:!say:))				80
input	hello		say hello				80
output	hello		Hello.				
phrase	please		please / plz				
phrase	say		say / tell me / repeat		command=say		

This bot has two *input filters*. The first one labelled *please filter* is the same as in the previous examples, it removes the word “please” or “plz”. The second filter labelled *say filter* is different because the phrase *((say))* has a *Learn*. It replaces the word “say”, “tell me” or “repeat” with “say”.

If you tell this bot “please tell me hello”, then the *input filter* labelled *please filter* first strips the word “please”. Next, the *input filter* labelled *say filter* matches the word “tell me” and learns the variable *command* as the word “say”. This value is then put back into the input string, replacing “tell me” with “say”. The input that is finally seen by the *input* labelled *hello* is simply “say hello”.

The same effect could have been achieved by just putting *((say)) hello* into the *input* labelled *hello*. But then I wouldn't have been able to show off this technique.

Replacement *input filters* are useful in complex bots which need to normalise the user input before processing it. They allow complex bits of processing to be done once in an *input filter* instead of lots of times in each individual *input*. The technique is also useful for *output filters*, shown in the next example.

Important points:

- *input filters* can also replace text
- any *phrases* with *Learns* within *input filters* replace rather than remove
- the replacement value is the value of the variable that was learned
- the name of the actual variable doesn't really matter unless you want to use it elsewhere

Example 7.4 – Output Filtering

Output filters work in the same way as *input filters*, except they operate on what the bot is just about to say. Just before a bot is about to deliver its *output*, it looks for any *output filters* belonging to the *output start* or to that *output* or anything it borrows from, and it processes them in the order they appear in the spreadsheet.

Output filtering is useful when combined with Clever Data, as in this example:

Type	Label	D	Text	If	Learn	Goto	Acc
output start	welcome		$clever_output$		clever_match=human		
output filter	human filter		((:!human))				75
output filter	name filter		((my name is)) ((!bot_name))				75
phrase	human		human / person		replace=monster		
phrase	my name is		my name is / I am				
phrase	bot_name		~		name=Evie		

This bot doesn't have any of its own *inputs* or real *outputs*. It is just a thin layer on top of Clever Data. The *output start* delivers the variable *clever_output* which comes straight from Clever Data. The variable *clever_match* means that it will favour statements containing the word “human”.

This bot has two *output filters*. The first catches any mention of the word “human” and replaces it with “monster”. It works just like the *input filter* from the previous example. It even has an *Accuracy*. The : means that the word being replaced has to start with “human”, so “humans” will be replaced by “monsters” but “semihuman” won't turn into “semimonster”. In output filtering, capitalisation is automatic if “monster” appears at the start of a sentence.

The second filter looks for things like “my name is ….” and replaces the name part with “Evie”. This shows another useful aspect of filtering – not everything in the filter needs to be filtered. In this case the “my name is” part will be left alone, and only the actual name will be replaced. This *output filter* is very useful if the Clever Data you are using happens to have learned a name for itself which you'd like to override.

You can test this bot by trying to get it to say the word “human”. For example, ask it “Are you a human or a robot?” The second *output filter* is much more difficult to trigger as the Clever Data shouldn't ever say it's own name. So alternatively, replace the *output start* with “I am a human” or “My name is Alan” to guarantee seeing the replacement filters in action.

Important points:

- *output filters* can remove or replace text in *outputs*
- if a replaced word appears at the beginning of a sentence, it is automatically capitalised
- *output filters* belonging to the *output start* are always processed
- *output filters* belonging to the *output* being given are also processed
- along with *output filters* belonging to any borrowed *outputs*
- *output filters* have an *Accuracy* and a *Special If*
- not all text in *input filters* and *output filters* has to be removed or replaced
- *output filters* can have *Ifs* and *Learns* which behave in the same was as *phrases*

Section 8 – Testing and Using Your Bot

The very first example in this manual briefly described how to turn your spreadsheet into a bot you could chat with on the www.cleverscript.com website.

This section expands upon that. It takes you through the steps of registering, uploading your spreadsheet, chatting to your bot, buying interaction credits, publishing your bot on our servers and communicating with your bot through our API.

8.1 - Register and Login

www.cleverscript.com is Cleverscript's home. It's where you can upload your spreadsheets and then chat with your newly created bots.

Registering

If not, please visit www.cleverscript.com and click the *Register* link in the right column, below the *Login* boxes. A small form will popup asking you to choose a username and enter your name and email address. We will then email you a password.

Login

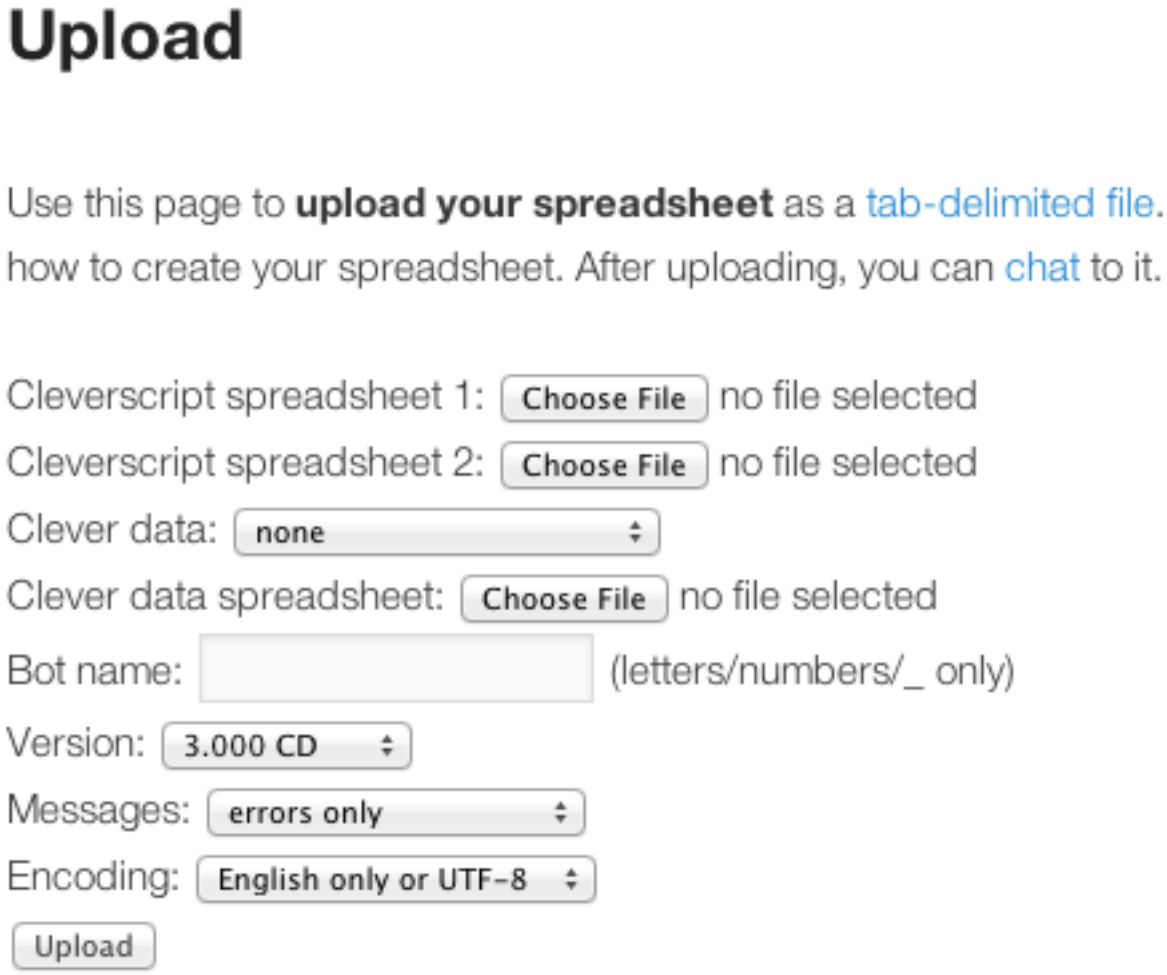
Once you've got your password, return to the website and login using the boxes on the right. You will arrive at the “Your bots” page. You now have access to the Upload, Chat, Buy Credits and Publish pages.

8.2 - Manual

This section is only here so that the section numbers correspond to the numbers on the Your Bots page. As you are currently reading it, there's not much more to say about the manual right now.

8.3 - Upload your Spreadsheet

The Upload page is where you upload your spreadsheet files. There are several boxes, all described below. The page looks something like this:



Saving your spreadsheet
The first step is to save your spreadsheet in **tab-delimited format**, which means the file name will end in tsv, txt or csv, and not xls. Here are instructions for some common spreadsheet applications:

Open Office 3 Calc

1. From the File menu, choose Save As
2. Change the **File Type** to **Text CSV**
3. Check the **Edit Filter Settings** box below that
4. Click Save
5. Choose Replace if it asks you
6. Choose Keep Current Format if it asks you
7. Under Character set choose UTF-8 if needed
8. Change the **Field delimiter** to **{Tab}**
9. Click OK

Microsoft Excel

1. From the File menu, choose Save As
2. Change the **Save As Type** to **Text (tab delimited)** or **Unicode** if it contains non-English characters
3. Click Save

4. If it asks about only saving the current worksheet click OK
5. If you saved it as Unicode, choose the **UTF-16** encoding when uploading

Google Docs Spreadsheet

1. From the Google Docs File menu, select **Download as** and then **Text (current sheet)**.
2. Your tab-delimited text file will be displayed in a new window, go to this window
3. From your browser's File menu, select Save As or Save File As
4. Name your file, specifying the file type as a Text File (*.txt) if possible
5. Click Save

Spreadsheets

The first two boxes are for uploading your spreadsheet(s). You can upload more than one, so you could keep a phrase library in one spreadsheet, and your inputs and outputs in another. Select the file (or files) you have created from your computer. As above, make sure they are in **tab-delimited format**. Any other format, like an Excel XLS file or comma-delimited text, will not work.

Clever Data

You can choose whether your bot should have some generic small talking ability, as described in the previous section. The options here are generally themed. The bigger databases are more intelligent but a bit slower. Note that the Cleverscript website uses much smaller versions of these databases than the API. So once you publish your bots, they will appear more intelligent.

Clever Data Spreadsheet

Advanced users can upload a spreadsheet containing their own Clever Data **instead** of using one of the provided choices. The spreadsheet should contain conversations. The *output* column is what the bot will say in response to a user's *input*. There are several different ways to arrange the spreadsheet columns:

1) *input*, *output*, *reaction* and *emotion*, where a blank *input* specifies the start of a new conversation
2) *conversation reference, line reference, context, input, output, reaction, emotion* where *context* is the statement that led to the *input* which led to *output*
3) *output* only where each *output* is used as the *input* for the next line in the spreadsheet and a blank line signifies the start of a new conversation (a single line by itself is ignored)
4) *output* only as above but only lines prefixed by "__" or "botname:" are used as actual *outputs*, which actually allows for a one column alternating format of *input* then *output*, the botname must (case insensitively) match the bot's name as specified under "name your bot"

If you would like to be able to use this facility, please let us know.

Name your bot

Choose a name for your bot. Only letters, numbers and underscores are allowed, no spaces or other punctation. Your customers will never see this bot name. It is for your and the website's reference only.

Version

The version field only appears if you have published bots using more than one version of the Cleverscript software. When we make major improvements to the software, we increase the version number. This allows bots created using a previous version to still run through the API. If you see this field, you should generally always choose the latest version.

Messages

The screenshot above was taken just after a spreadsheet was uploaded. It shows that there was an error in one of the Gotos. These messages are very useful for debugging. There are various levels of

messages including just errors, or warnings, statistics and timing too. If your bot does not work as intended, this is the first place to check.

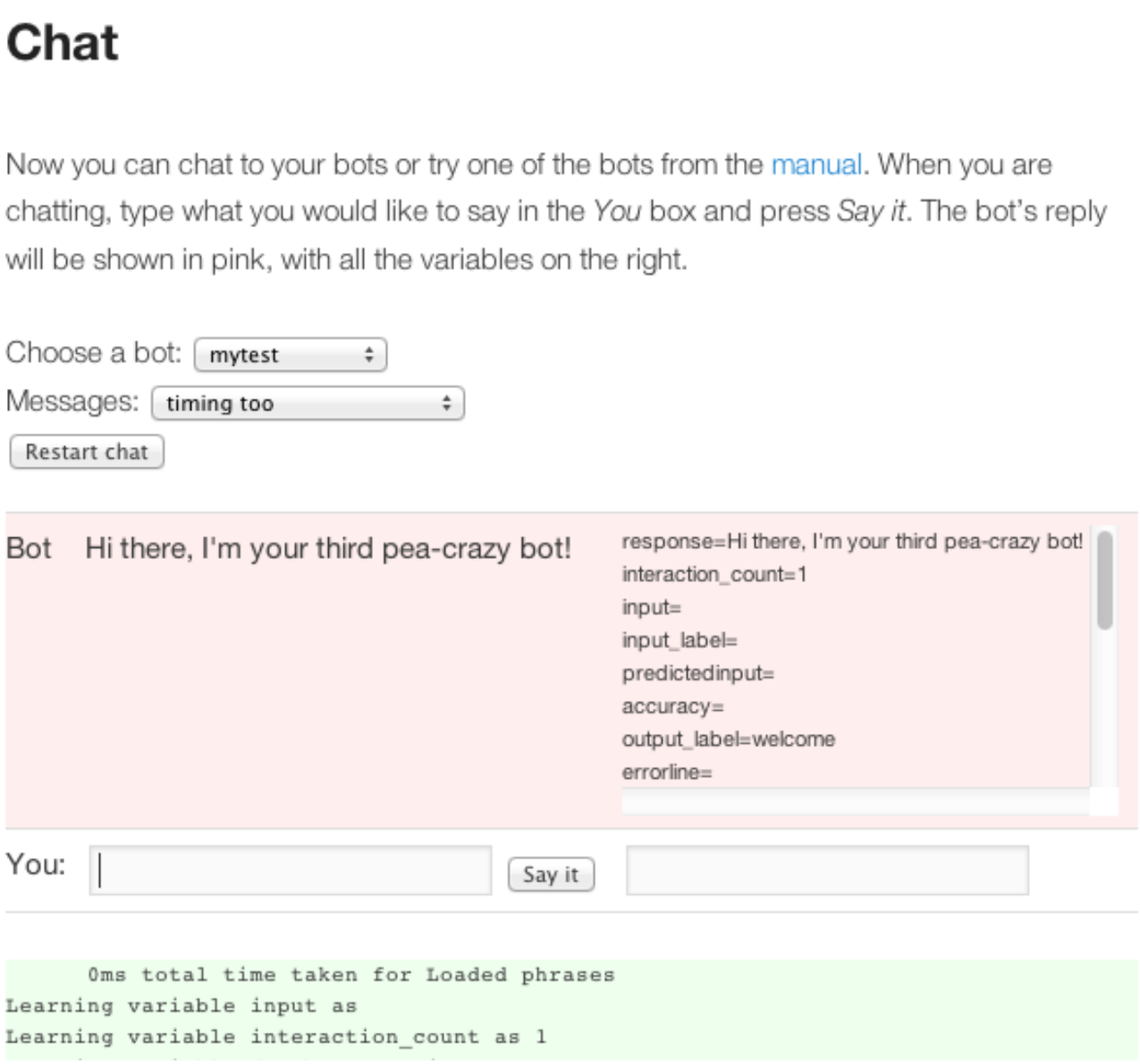
Character Encoding
Cleverscript works only with UTF-8 text. If your bot contains only English numbers and letters, then it is already compatible (as ASCII is a subset of UTF-8) and you don't need to worry about this. If it contains any non-Latin characters, like Greek or Cyrillic or accented letters, then you should try to convert your spreadsheet into UTF-8 before uploading.

However, we're aware that spreadsheet software like Excel can make it very difficult to save in UTF-8 format, so we are able to convert from a limited number of other encodings. Excel's Unicode option saves as UTF-16 so you can convert from this. Or else, you have to know what encoding you are using. If your favourite encoding is not in the list, please let us know and we will add it. Note that this may not work in older browsers and you may have to upload in UTF-8 anyway.

Upload
Press Upload when you're ready. It will display any error messages (if you wanted them) and provide a link to chat with your bot.

8.4 – Chat to Your Bot

After you have successfully uploaded your bot, you can chat to it. The chatting page looks like this:



Choose a bot
If you click the "chat" link straight from the Upload page, your bot will start chatting to you right away. Otherwise, choose your bot from the list.

Messages
There are also various levels of messages when chatting to your bot. This ranges from no messages at all to a very detailed output listing all inputs it tries to match against and all the variables learned along the way.

Start at output label
Advanced users may see this optional box, which allows you to enter an alternate output label (rather than the *output start*) for the chat to start at.

Start chat
Press the “Start chat” button to start a chat. If a chat is already in progress, the button will say “Restart chat”.

Your conversation
During a chat, your bots statements are shown in pink and yours in white. On the right of the pink boxes is a list of all the variables within your bot at that moment in the conversation. This list starts with the many system variables and has your bot's own variables after that. For example, the variable *interaction_count* starts at 1 and will increase as the chat progresses.

Type what you would like to say next to the “You” on the bottom left. On the bottom right is a box for overriding variables. For example, if you have a variable called *colour* you could type *colour=green* into this box. To set multiple variables, use the same format as in the spreadsheets: *colour=green and size=large*.

Press “Say it” or the Enter key to say something. The page will refresh and your bot's reply will be added to the bottom of the conversation.

8.5 – Buy Credits

After you have created your bot, you will probably want to use it somewhere. Cleverscript bots can be used in websites, SmartPhone apps, console games and computer games.

Apps and games
For SmartPhone apps, console games and computer games, we will provide the Cleverscript softare as a library which you can include in your project. Your bot will then be downloaded from the Cleverscript website. Cilck to the next section about publishing to find out how to do that.

Website usage
To use your bot on your website, we provide a JSON API. With a few lines of Javascript, your website will communicate over the Internet with our servers, sending your bot some input and getting the reply.

Every user input and reply is called an *interaction*. When you are having a conversation, this is represented by the internal variable *interaction_count*.

Interactions credits
We charge per interaction. Interactions must be bought in advance from the *Buy Credits* page of our website. Current pricing is available on the prices page on our website. Interactions are sold in preset packages. Our website currently integrates with PayPal though we will add other providers soon. All prices are in US dollars.

Extra credits
We provide 1000 extra credits as part of your first purchase. This is to cover any credits you may use in integrating our API.

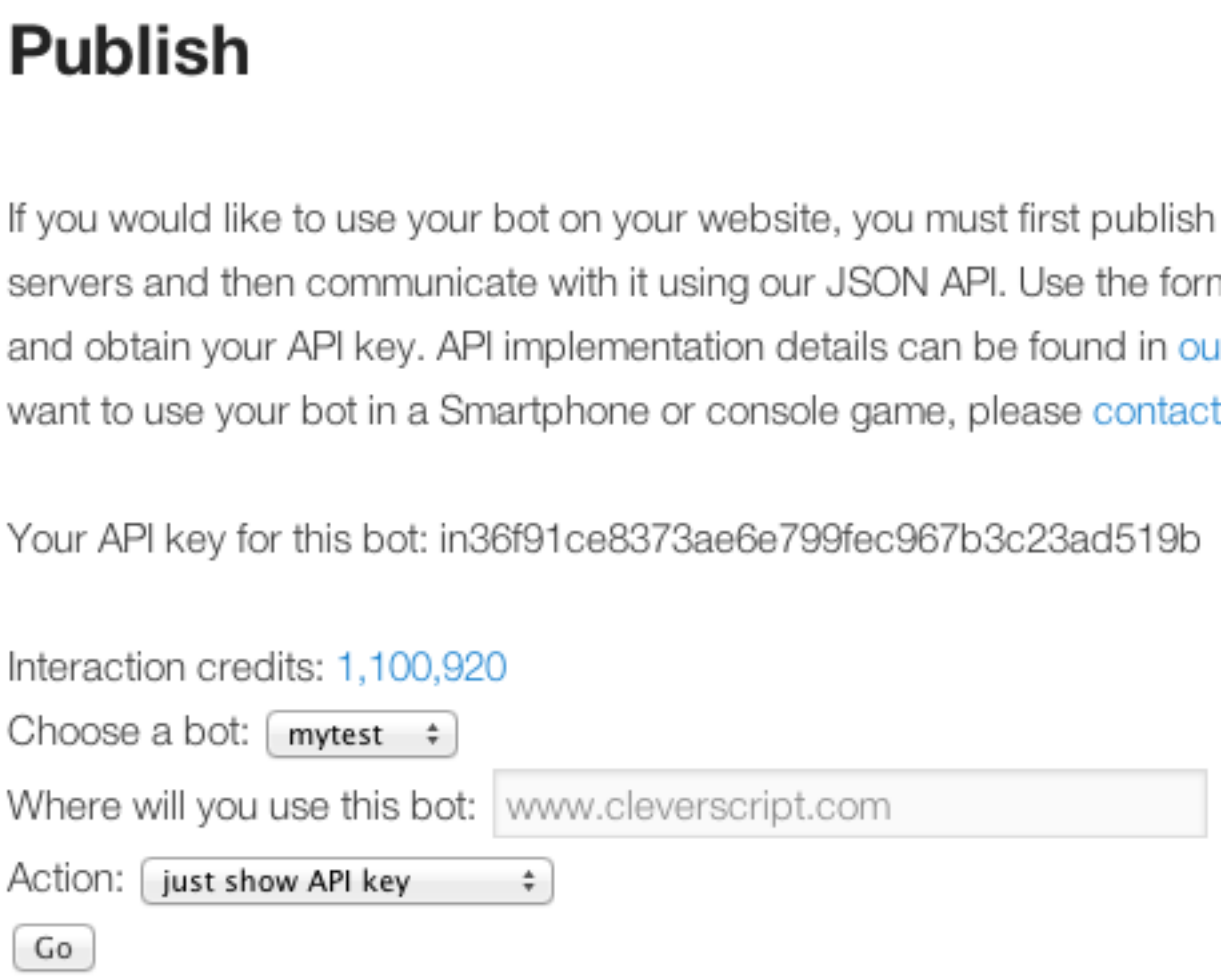
Buy credits
Decide the quantity you would like and press the Pay button. You will be forwarded to our payment provider to take payment. After a successful payment, you can return to our website and visit the Publish page.

Low credits
You can visit our website and top up your credits at any time. We will send your an automated reminder email when your credits are running low.

If your credits run out, your bot may return an error message to your users, something like “This Cleverscript bot can not be used from this website.” We will try to contact you before taking this step.

8.6 – Publish

When you are ready to make your bot live and available, so that your customers can chat to it, visit the publishing page. For website usage, this page will publish your bot to our servers and give you an API key. For app and console usage, we will provide a file to embed in your project. The publish page looks like:



Interaction credits
The publishing form first tells you how many interaction credits you have remaining. If you have any live bots, the number will be a link to a statistics page (discussed below).

Choose a bot
As on the chatting page, you can choose the bot you would like to publish.

Where will you use this bot
If you are connecting to our JSON API from Javascript, then enter the domain name of the website where your bot will be used, such as www.cleverscript.com. If you will connect to our API directly from a server using a language like PHP, then enter the IP address, such as 123.123.123.123. Both methods are explained in the API section. Note that **your bot will only work from this address**. And if you change this address, **your API key will also change**.

Action
The action field has several options:

Download file

If you have told us that you will be using your bot in an app or game, then the “download file” option will appear first. This will give you a link to directly download your bot's file. The file is essentially a database which you can embed into your application.

Just show API key
Choose this option to see your current API key for this bot. Each bot will have a separate API key. As above, the API key will change if you put your bot on a new website.

Publish to live servers
Publishing involves copying your bot's file to our servers. The website will report if the publishing was successful and how many servers it was published to. We have several load-balanced servers to make sure your bots always respond quickly. The same API key will work on both sets of servers. You will only see this publish option if you have purchased interaction credits.

Change API key
Use this option to change your API key. After changing the key **you must republish** for it to take effect. Once you have republished, **your old API key will no longer work**.

Unpublish from servers
If you would like to remove your bot from our servers, choose this option. It will remove it from the live servers. You will have to republish to get it to work again.

Delete bot
This will delete your bot from your “choose a bot” list and unpublish it from our servers.

8.7 Javascript API

We provide a JSON API for interacting with your bot over the Internet. There are two main ways to use this API: in the browser using Javascript, or on a server using a language like PHP. You can obtain your API key on the Publish page of www.cleverscript.com. This section shows how to set up your own Javascript API chat, but we strongly recommend using our library at www.cleverscript.com/CSL/cleverscriptapi.js.

The Quickest Way

When Publishing, enter your website's domain for the "where will you this bot" question. Then in your web page, you just have to add 2 lines of HTML to get a fully working conversational bot:

```
<script type="text/javascript"
src="http://www.cleverscript.com/CSL/cleverscriptapi.js"></script>
<script type="text/javascript">ShowCleverscriptForm ('YOUR_API_KEY');</script>
```

This Javascript library that does it all for you – including creating a form and fading between reaction and emotion images if you are using Clever Data. The library also handles very long URLs, which can happen if your bot has lots of variables which give it a long state variable. URLs over 2048 characters don't work in some versions of Internet Explorer. The library fixes this for IE8 and above by using window.postMessage.

Occasionally this library is modified, so you can refer directly to the copy on our servers if you would always like the latest version. However, we can not guarantee that the format will always stay the same (though we will try to make it as backward compatible as possible) so you may want to copy the library and use from your own servers.

With Your Own Form

You can also use our library with your own form, by calling *CleverscriptSetup* and *CleverscriptInput* and using your own callback function, like this:

```
<form onsubmit="CleverscriptInput (this.userinput.value); return false;">
You: <input name="userinput" type="text"/></form>
<script type="text/javascript" src="/path/to/cleverscriptapi.js"></script>
<script type="text/javascript">
function mycallback (data) {alert ('Bot replied: ' + data.output);}
CleverscriptSetup ('YOUR_API_KEY', {callback_after:mycallback});
</script>
```

Do-it-yourself Javascript chat

You don't have to use our library. Implementing a basic version of our API yourself is easy. As with other JSON APIs, it involves inserting a *<script>* tag into your web page with a Javascript callback. For example:

```
<script type="text/javascript">function CSProcess (data) {alert (data.output);}</script> <script
type="text/javascript" src="http://api.cleverscript.com/csapi?
key=YOUR_API_KEY&input=Hello&callback=CSProcess"></script>
```

You can copy the code above into an HTML file on your server. Insert the API key for your bot into the URL and view the page in your browser. It will only work from the domain name you specified during the publishing process.

The URL for our API is *http://**api**.cleverscript.com/csapi*.

If successful, the above code should call the *CSProcess* function which will cause a Javascript alert with your bot's response to the input “Hello”. If you try to view this code from a different website, you will get a response like “This Cleverscript bot can not be used from this website.”

The data object
The callback function receives an object as its only argument. The object contains all the variables from the bot's response. The variables are the same as the ones in the pink box on the *Chat* page. You can show all those variables by changing the *CSProcess* function to:
function CSProcess (data) {var r=''; for (var p in data) r += p + ': ' + data[p] + '\n'; alert (r);}

You can see that the *output* property is the first and most important bit of data, but you also have access to all your bots variables. The second most important one is the *cs* variable. That contains Cleverscript's internal state, including where it is in its spreadsheet and the value of all its variables. You must pass this *cs* variable back into the API the next time you call it.

So your next call the API will look something like:
<script type="text/javascript" src="http://api.cleverscript.com/csapi?
key=YOUR_API_KEY&input=Hi&cs=YOUR_CS_VARIABLE&callback=CSProcess"></script>

Overriding variables
As on the Chat page, you can also override variables in the API. To override a variable, prefix it with an underscore. For instance, if you want to force the variable *$colour$* in your script to have the value *red,* then use:
<script type="text/javascript" src="http://aiapi.cleverscript.com/csapi?
key=YOUR_API_KEY&input=Hellooooooooo&cs=YOUR_CS_VARIABLE&callback=CSProcess
&_colour=red"></script>

This is also very useful if you'd like your bot to start somewhere other than the *output start*. You can pass in *output_label=other_place* to override the internal *$output_label$* variable. This is also built into our Javascript library as an argument to *ShowCleverscriptForm* (for the first interaction) and *CleverscriptInput* for later interactions.. See the Javascript file for how to do it.

Complete chatting bot
Below is HTML and Javascript for making a complete do-it-yourself Javascript chatting bot. Save this to an HTML page on your server to try it out. It has two Javascript functions: *CSInput* takes the user input and constructs a *<script>* tag like the ones above. It then appends it to the * * at the top of the page, which causes the *<script>* tag to be interpretted by the browser, calling the API. *CSProcess* then receives the data and adds your input and the bot's reply to the same * *.

```
<div id="reply"></div>
<form onsubmit="CSInput (this.userinput.value); this.userinput.value = ''; return false;">
You: <input name="userinput" type="text" size="40"/> <input type="submit" value="Say it" />
</form>

<script type="text/javascript">
var CSSTATE = '';
function CSProcess (data) {
   if (data.input) document.getElementById ('reply').innerHTML += 'You: '+data.input+'<br/>';
   document.getElementById ('reply').innerHTML += 'Bot: ' + data.output + '<br/>';
   CSSTATE = data.cs;
}
function CSInput (userinput) {
   var key = 'YOUR_API_KEY';
```

```
    var url = 'http://api.cleverscript.com/csapi';
    url += '?key=' + key + '&input=' + encodeURIComponent (userinput);
    url += '&cs=' + CSSTATE + '&callback=CSProcess';
    var el = document.createElement ('script');
    el.setAttribute ('action', 'text/javascript'); el.setAttribute ('src', url);
    document.getElementById('reply').appendChild (el);
}
CSInput ('');
</script>
```

When you first load the page, the function *CSInput* is called with a blank string. This causes the API to get the bot's opening statement. After that the global variable *CSSTATE* is used to store the Cleverscript state between calls. The state is saved in the *CSProcess* function and used again in *CSInput.*

Server-side chat

You can also use our JSON API directly from your server. In this case, you have to enter your server's IP address into the “where will you use this bot” box, and leave off the callback when you make the API call. The API will then return a raw JSON variable which can be parsed. Here is an example in PHP. This will retrieve the data, decode it using the PHP function *json_decode* and display the results:

```
<?php
  $data = file_get_contents ('http://api.cleverscript.com/csapi?
key=YOUR_API_KEY&input=hello&cs=YOUR_CS_VARIALBE');
  var_dump (json_decode ($data, true));
?>
```

Animated Avatar

From April 2014, you can use our animated avatar with your Cleverscript bot. This is the same avatar which appears at www.existor.com and in several very popular YouTube videos.

To use the avatar on your website, you need to contact us first and send us your website address. If approved, we will send you the SWF file you need. Then you simply pass in the name of this SWF file into ShowCleverscriptForm function.

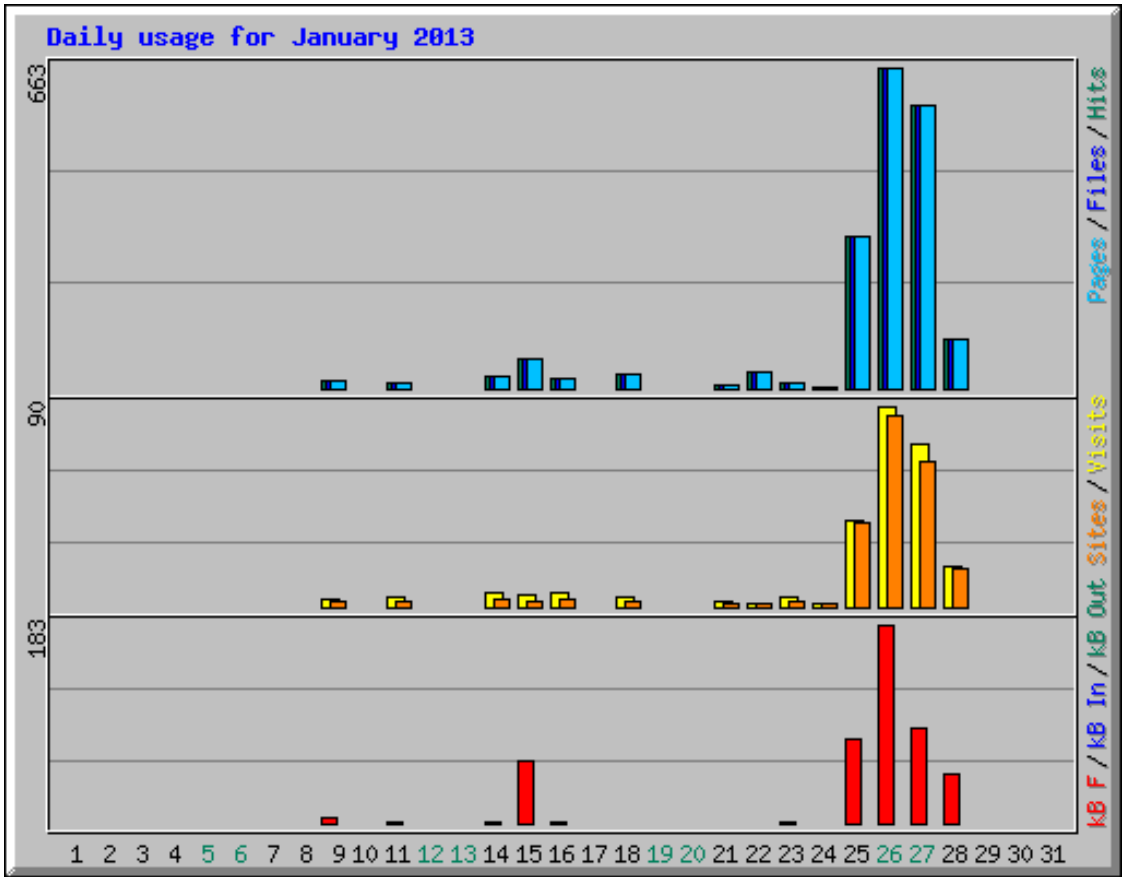
Note that each interaction with an avatar costs more than 1 credit. See the prices page for current pricing.

8.8 - Statistics

We provide some statistics about your Cleverscript usage on our servers. These are only available 24 hours after you first publish to our servers and people start chatting with your bots. This only applies to website/Internet usage, not app/console usage.

There is a general stats file accessed from the Publish page (click the number of credits you have left) or the Statistics page. Each bot also has its own stats file, accessible form the Statistics page. Stats are compiled at midnight GMT.

Statistics are shown using the Webalizer package. They include monthly, daily, hourly, country/domain and other common website stats. Below is a sample of daily usage stats.



These stats are provided as a monitoring device, so you can see where your credits are being used. We encourage you to keep your own stats as well.

8.9 – Android and iOS APIs

Our Android and iOS APIs allow you to embed your bot into an Android or iOS app. They are very different from the Javascript API as that mostly deals with HTML and forms. They are however very similar to each other, so they are presented together, with different font colours for Android or iOS specific instructions.

Tell us

First you need to tell us you would like to use your bot in an Android or iOS app, and agree to our licensing and costing. We will then send you library files for including in your app, and enable you to download the necessary database files from the Publish page as above.

Android instructions

We will send you two files: a library ending in .so and a .jar file for the headers.

- Create the /libs/ and /libs/armeabi/ directories if not there
- Copy cleverscriptapi.jar into /libs/
- Copy libcleverscriptapi.so into /libs/armeabi/
- Add *import com.cleverscript.android.*;* to your Java
- Run initial test (below)
- Visit the Publish page, enter your package name and download your bot's database files
- Copy db file into /assets/mybot.db.jpg
- Copy dbe file into /assets/mybot.dbe.jpg

iOS instructions

We will send you three files: a library for the simulator (libCleverscript-simulator.a including i386 and x64_86 architectures), a library for the device (libCleverscript-device.a including armv7, armv7s and arm64 architectures) and a header file (CleverscriptAPI.h):

- Right click the project name (eg MyApp) at the top of Navigator area on the left and choose "Add Files to MyApp"
- Add all the files we sent you – both libraries and the header file. Note that when you release the app, you can remove the libCleverscriptAPI-simulator.a library; it is just for testing on the simulator. When compiling with both you will probably get a warning about missing architectures - this is okay, it's because the simulator doesn't recognise the device's library and vica versa.
- Click the project name, select the appropriate Target (in the middle panel of XCode), select the Build Phases tab, open the "Link Binary With Libraries" section and click +
- Add libstc++.dylib and libsqlite3.dylib to enable your project to use our C++ library and your bot's SQLite database
- You may want to move the libraries and dylib files into the Frameworks folder or to a new group to keep your files organised.
- Add *import "CleverscriptAPI.h";* to the top of your main App delegate source file
- Run initial test (below)
- Visit the Publish page, enter your package name and download your bot's database files
- Add the db and dbe files to your app

Initial test

After adding the library, you can run an initial test to make sure the API works by outputting a reply from the API to the log.

For Android, you must pass the constructor your application's context. The output goes to LogCat:

```
CleverscriptAPI cs = new CleverscriptAPI (getApplicationContext());
Log.i ("CSANDROID", "From API: " + cs.sendMessage ("hello world!"));
```

For iOS, add the following to your app's *didFinishLaunchingWithOptions* method or somewhere similar. The output goes to the NSLog:
*CleverscriptAPI *cs = [[CleverscriptAPI alloc] init];*
NSLog (@"From API: %@", [cs sendMessage:@"hello world!"]);

If successful it should say “From API: no database loaded” in the log.

Prepare to download your bot
To test your API for real, you will need to first obtain your bot's database files and API key. To do this, go to the Publish page on the Cleverscript website and choose your bot. The “where will you use this bot?” question is important as your API key won't work unless it is correct:

For Android, enter your package name. This appears at the top your main Java file, something like “com.example.myapp”.

For iOS, enter your app name. To find this, click the project name, select the appropriate Target, click Summary and look for the Bundle Identifier. This will be something like “com.example.myapp.MyAPP”.

Change the Action to “download database” and press “Go”. The page will refresh.

Database files and API key
Click the “download your bot's database” link and save the file onto your computer (should end in .db). Do the same for the “helper file” (should end in .dbe). The helper file is a memory dump which helps your database load much more quickly. This page will also give you your API key.

For **Android**, copy your database and helper files into the **/assets/** directory of your app. If either file is over 1Mb (which it will be if you have included any Clever Data), then you must add “.jpg” to the end of each file. For example, **rename mybot.db to mybot.db.jpg** and **mybot.dbe to mybot.dbe.jpg**. This is the easiest way of telling Android not to try to compress the files (which is limited to files up to 1Mb). Do not otherwise change the filename of your bot as it is also tied to your API key. Now use the *setLocation* function to tell the API where your database is. For Android, the location is relative to the /assets/ folder, so leave off the /assets/ and the .jpg is optional:
cs.setLocation ("mybot.db");

Android apps are written to one large .apk file. So before your bot's database can be used it is copied out of the /assets/ part of the .apk file and placed into local internal starge. If you have external storage available, you can bypass this by passing in an absolute path starting with / such as /mnt/sdcard/myapp/mybot.db.

For **iOS**, add the db and dbe file together to your app. Then call *setLocation*. The location must be a full path name, such as:
NSString filePath = [[NSBundle mainBundle] pathForResource:@"mybot" ofType:@"db"];*
[cs setLocation:filePath];

If you run your app with just that, your bot should reply “invalid API key”. So you should also call *setApiKey* with the API key from the previous step. Then *sendMessage* should return a proper reply.

In real use, you probably want to call *loadDatabase* separately from *sendMessage* when your app first loads. And both should be run asynchronously in the background as they can take a few hundred milliseconds or more.

Debugging
If you call *setDebugLevel(1)* the Cleverscript API will send debugging messages to the log. Higher numbers give more debugging information from inside the API. In Android, it goes into the "info" log of LogCat with the tag CSANDROID. iOS adds to the general NSLog.

The bot's reply also contains minimal error reporting (for database and API key problems). The *loadDatabase* function can be called separately from *sendMessage* and has a return value which gives some error information as described below.

API functions
The API provides several other methods for getting and setting Cleverscript variables, and saving and restoring conversations. Here is a full list of what the API can do. For iOS the API accepts NSString* rather than String.

- *new CleverscriptAPI (Context yourcontext)*: For Android only, you must pass in your application's context to the Cleverscript API constructor.
- *void getVersion()*: Returns an integer version number of the API.
- *void setDebugLevel (int level)*: Pass in a debug level between 0 and 4 to see various error messages. These are mostly the same messages that you can see while chatting to your bot on the Cleverscript website.
- *void setLocation (String location)*: Specify the location of your database. For Android, leave off the /assets/ and optionally the .jpg. For iOS pass in the full path name from pathForResource.
- *void setApiKey (String apikey)*: Specify your bot's API key (about 30 characters long).
- *int loadDatabase()*: this extracts the db and dbe files from the app's apk file, puts them into internal storage, and then loads their contents into memory. Note that your bot will still work without the dbe file, but the loading will take much longer. It returns:
  - 0: successful
  - -1000: no database location provided
  - -1001: could not find or open database db file (check you added .jpg to the file names)
  - -1002: could not copy db file to internal storage
  - any other number < 0: internal error, the bot's reply should give more details
- *void unloadDatabase()*: The Cleverscript API loads all data into memory, this unloads it all.
- *String sendMessage (String input)*: If the database has not already been loaded, then this will call *loadDatabase* and then send the input to your bot and return the reply. If the database has not been specified, has the wrong API key, or has not loaded correctly, then the reply will contain an error message.
- *void clearConversation()*: The Cleverscript API keeps its own internal log of the conversation and uses it to influence Clever Data replies contextually. This clears the log.
- *void addInteraction (String user, String bot)*: Once cleared, you can load in a previously saved conversation line-by-line with this function, providing each user input and bot reply. Though a better way to load old conversations is with *assignBotState*.
- *String retrieveVariable (String variableName)*: Returns the requested Cleverscript variable. If the variable has never been defined inside your bot and is not dynamic like *myvariable{myindex}*, this will return null. Variables are defined by appearing at least once in the *If* or *Learn* columns.
- *void assignVariable (String variableName, String variableValue)*: Assigns a value to a variable within Cleverscript. Note that variable values are not stored permanently. They are only kept in memory while the app is running and will be reset whenever the app restarts. As with *retrieveVariable*, you can also only assign variables which have been previously defined within your bot, or are dynamic.

- *String retrieveBotState()*: Returns a string containing the state of the bot, which is an encoded version of all the variables and the conversation history. This can be saved and later given to *assignBotState* (including the *cs*=) to restart a previous conversation.
- *void assignBotState (state)*: Used to restart a previous conversation from a state string previously retrieved with *retrieveBotState*. You can also pass in variable overrides by adding things like *myvar=value*, with newlines between each.
- *String retreiveSessionId()*: Returns the Cleverscript session ID variable, which is string about 10 characters long, randomly created at the beginning of a conversation.

Two extra functions are available for iOS only from October 2014:

- *void setDebugFile (String location)*: Outputs all debugging messages to the given file.
- *void setMemoryLimitMB (int megabytes)*: Set a limit on the approximate number of megabytes that the API can use. Applies only when loading Clever Data.

Emotional Data

We can also enable emotion functionality in your API and database. This will provide you with the following additional functions:

- *void sendMessageForExpressionValues (String input)*: Just compute the expression values (reaction and emotion) for a given input, without replying. You can get the emotion data back out by retrieving Cleverscript variables, such as *retrieveVariable("emotion_values")*.
- *void matchExpressionValues (String input)*: Pass in a comma separated list of emotion values such as 0,0,0,0,100,0,0. The numbers refer to the seven basic emotions: anger, fear, disgust, contempt, joy, sadness, surprise. This also works with the reactions and emotions built into Cleverscript such as “happy”. Call this before calling *sendMessage* to influence the bot's reply.

Here is an example of using this functionality in Android to get the emotional values for a piece of text. The variable *emotions* will be a string representing the seven basic emotions such as “0,0,0,0,50,0,0”. Please note that guessing emotions from text is imprecise and approximate:

```
cs.sendMessageForExpressionValues ("I am going to eat lunch.");
String emotions = cs.retrieveVariable ("emotion_values");
```

And here is a corresponding example for influencing the bot's reply by passing in basic emotion values. This will favour an angry reply. Please note that influencing the reply is also approximate as there are many other factors affecting it as well:

```
cs.matchExpressionValues ("100,0,0,0,0,0,0");
String reply = cs.sendMessage ("How are you doing?");
```

8.10 – Self Hosting

Subject to a licensing agreement and pre-payment, you can host our software yourself. Once agreed, we will send you an executable which will run as a mini web server and accept and respond to JSON requests. When downloading your bot's database, you will need to answer the question "Where will you use this bot" with "*self-hosted*".

For example, in Linux you can run it as:
CleverscriptSelfHostedWebServer.out mybot.db -key YOUR_API_KEY -wp 9000

This starts a web server on port 9000 associated with the database file and API key retrieved from the Publish page of the Cleverscript website.

You can then communicate with your bot using the same JSON techniques as described above. You have to make the request to your own server on the given port. Request the URL /json followed by a ?. For example, making a request to:
http://myserver.com:9000/json?key=YOUR_API_KEY
&cs=YOUR_CS_VARIABLE&input=Hi&callback=CSProcess

Will return something like:
CSProcess ({"input": "Hi", ...});

To call this directly from Javascript:
<script type="text/javascript" src="http://myserver.com:9000/json?key=YOUR_API_KEY
&cs=YOUR_CS_VARIABLE&input=Hi&callback=CSProcess"></script>

Or you can use our Javascript library. Note that you don't have to pass in an API key.
<script type="text/javascript"
src="http://www.cleverscript.com/CSL/cleverscriptapi.js"></script>
<script type="text/javascript">ShowCleverscriptForm ('', {server:'http://myserver.com:9000/json', debug:2});</script>

The web server can also reply with XML if you call the URL:
http://myserver.com:9000/xml?cs=YOUR_CS_VARIABLE&input=Hi&callback=CSProcess

Once running the web server can only be shutdown nicely by visiting:
http://myserver.com:9000/shutdown?botname=mybot&key=YOUR_API_KEY

Please note that other than the API key check, anybody could connect to this URL and chat to your bot.