Context is treated as a long segmented text, composed first from the segments that come in from the user and second from the responses generated by the chatbot – as I’ll explain.

This long segmented text can be read in reverse to “grab” desired types of VAR when triggered by a word like “both”, or “difference”, or “it”. Such words are a special kind of VAR. And the types of things that can be grabbed are also another special kind of VAR.

How Context VARs work

VARs now have two new fields: contextFn and contextType.

VARs with non-trivial **self.contextFn** are able to extract old VARs from a (context) segment. VARs whose *parents* have a non-trivial **self.contextType** are, by definition, extractable from the context. The type involved is closely linked to the contextFn that can grab them.

For example: the **DIFF** VAR has a contextFn called get2Alternatives() that returns a pair of alternatives from the recent context. Those are VARs whose parent has the type ALTERNATIVE\_CONTEXT. So contextType determines which kind of VAR can be grabbed.

|  |  |  |  |  |
| --- | --- | --- | --- | --- |
| **VAR with contextFn** | **contextFn** | **contextType** | **example** |  |
| DIFF | get2Altermatives | ALTERNATIVE\_CONTEXT | Ti versdus Zr |  |
| BOTH | get2Mergablable | MERGE\_CONTEXT | Crown/Abutment |  |
| THOSE | getManyGroupable | GROUPABLE\_CONTEXT | Tooth numbers |  |
| IT | getOneGroupable | GROUPABLE\_CONTEXT | any object |  |

So the way a context function extracts is by returning a list of VARs from its contextFn applied to the saved context.

It is confusing because the NWChat object owns numerous topics – and each topic has its own copy of a saved context. Each topic’s context is, first, a record of the segments it created to process the text. But then VARs *from the container* are inserted into each topic’s context. These container VARs are called NWChat.responseVARs and they are set by the same mechanism that sets the response. The topics are decoupled from the response but the same responseVARs are inserted into each separate topic’s version of the context.

The NWChat.**RespondNext**() will be overriden in a particular chatbot to fill the self.responseVARs, but in the base class method NWChat.**respondeNext**(), those VARs are transferred back to the topics. Let’s look at the details:

When a topic does a read() it appends the new segment to its internal context. It will also get context from the chatbot response. Say the chatbot has a response of the form “blah blah titanium blah blah zirconia…” and responseVARs of the form [TITN, ZIRC].

Now In the base class NWChat.respondNext() we have

respondNext( ):

self.responseVARs = []

response = RespondNext() # get a text response from the derived class of NWChat

# and create content in self.responseVARs

for all topics in self:

topic.context.extend( self.responseVARs)

That is how the NWChat is responsible for owning the responseVARs and hands them back to each topic, for addition to that topics context. That is how a context derived from the response is added to context from the user input.

Later, when a topic does another read() and finds a score of GOF<=0.5, it goes into a recovery cycle:

* did the input contain a VAR with a non-trivial contextFn?
* if yes and it is applied to the context (including earlier responseVARs) and VARs are extracted
* and those VARs are inserted into a new segment
* try reading the new segement.

So: the first read() had a response that includes a VAR list (responseVARs). These are added to the previous segmented text that was appended to the ‘context’ of each topic in a chatbot.

A susequent read() has an inadequate GOF, so it goes into a mode looking for contextFn owning VARs and uses them to extract recent content from the context, in a second pass attempt to get a better GOF.

