Narwhal system files

This is a summary of file contents for the \narwhal\ directory. To study the code “low-to-high” begin with the data types in nwtypes.py. To study it “high-to-low” begin with one of the sample chatbots, like ‘AboutChat’. The Narwhal files reflect the Narwhal spec (See GitHub /peterwaksman/Narwhal/).

Overview

Rather than listing alphabetically, these are ordered (roughly) by dependency. Lower numbers have fewer dependencies. To study the code, begin with lower numbed files.

**(1) nwutils.py** – odds and ends (without definition dependencies)

**(2) nwtypes.py** – defines the principal narwhal data types. These are KList (list of keywords), VARs (concepts that are defined by Klists), and NAR (narrative pattern expressing a relation between VARs).

**(3) nwfind.py** – contains the findInText() method and sub-routines for matching VAR patterns to tokens.

**(4) nwcontrol.py** – low level narwhal, defines the operator trees, and the ControlData structure used by the narwhal “outer loop” in nwnreader.py.

**(5) nwsegment.py** – defines utilities around “segmented text”, and the narwhal “inner loop” read methods

**(6) nwnreader.py** – the reader class (actually the last one standing); implement the main “outer loop”

**(7) nwvault.py** – utilities used in the “outer loop”, concerned with archiving (“vaulting”) results during an extended read operation. Also contains recordSlotEvents(), another way to archive results.

**(8) nwchat.py** – chatbot class definitions

**(9) nwcontext.py** – utilities and VARs for reading values stored in context buffers

**(10) nwlog.py** – a simple data logger

Generally, low level data types are defined in nwtypes.py and token matching is in nwfind.py. These are used in reading functions in nwnreader.py and nwsegement.py. These are used by the app level clients. Currently I deleted all clients except the chat clients.

File details

nwutils.py

Odds and ends of low level utilities.

nwtypes.py

Defines the important narwhal data types: KList, VAR, and NAR.

**KList** – a wrapper around a comma separated list of “synonym” keywords. The wrapper performs token matching within the findInText() method. A KList is a global constant and, for example, can be retrieved from its name using the ‘instances’ method with this syntax: klist = KList.instances[name].

Note that the token matching in findInText() relies on use of special characters “\*”, “#”, “$”, “|”. These are used to disambiguate homonyms. “\*” to skip a token; “#” to exclude tokens after the match; “$” to exclude tokens before the match; and “|” to join excluded tokens. For example, suppose you want one var for the word “ready”, another for the phrase “ready to”, and another for the phrase “ready for”. The KList to use for the first contains “ready # for|to” – where ‘for’ and ‘to’ are excluded, the second contains “ready to” and the last uses “ready for”. See detailed commends in nwfind.py.

**VAR** – Initialized from a KList, a VAR is a “concept” defined by its keywords. VARs are connected to each other in a tree structure. To make var2 a subnode below the node of var1, use the syntax var1.sub(var2). A typical example is to put a ‘color’ VAR as a parent of a ‘red’ VAR. So you might expect to see both COLOR and RED defined by keywords and have the expression COLOR.sub(RED) to make RED a subnode COLOR.

VAR trees are important for text matching. Generally we seek the most detailed (“lowest level”) match within a tree – starting from the tree’s root node (the “highest level”) and following each branch as far as possible during the search. Since keywords are expected to be non-overlapping lists, any match will occur at most once. [Cycles and overlapped keyword lists are legal and the only downside (currently) is index confusion in one place – which contains a workaround. So you can do it but it may lead to some confusion.

By convention, a match is considered ‘true’ for all parents of the lowest level matched node. VARs contain a field called ‘foundInChildren’ to indicate when a VAR is a true match versus the parent of a true match. So where a VAR is to be matched, it is the lowest level child that will be found to match. Hence the higher level node is considered a “variable”, when it is used in the expression with the intention of being replaced by a child of the explicit VAR. For example the root node of the tree behaves like a variable when it appears in an expression and can be replaced by any node below it.

Note the implementation ‘<=’. The expression var2 <=var1 means there is a sequence of VARs, each a subnode of the previous, and connecting var2 below var1. In affect this means that var1 can behave like a variable whose value (for a given match) is set equal to var2.

*Special Types of VAR*

**Unkown** VARs. VARs are of two kinds: *fixed* value versus *unknown* value. Most VARs are of fixed values, as represented by the lowest level match described above. For such fixed value VARs, when a match occurs, it retrieves the VAR itself, named as it is named. A VAR with an unknown value is special kind of VAR that satisfies **isUnknown()** and is used to retrieve a string littoral from the text. We can retrieve ints and floats this way (see INTx and FLOATx), not as a matched concept but as a string littoral. We can also retrieve prefixed and suffixed strings. The design is in place to allow retrieving any regular expression, but this is not implemented.

Vars with a **context function**. Most VARs are created using a syntax like

v = KList(“name”, “synonym1, synonym2, …).var().

However, the construction allows passing a function name as an argument, like this

v = KList(“name”, “synonym1, synonym2, …).var(**contextFn**).

This ‘contextFn’ is stored inside the VAR and, when the VAR is matched in an *otherwise incomplete multi-part match*, the function can be called on a list of previous VARs (called the “context”) and try to retrieve the missing parts of the match. For example DIFF is a var which, when found in an incomplete multi-part match, allows retrieving the last two mentioned VARs of the desired type to complete the match. Examples are BOTH, DIFF, IT.

**NAR** – this is a simple narrative relation between VARs or other NARs and contains four parts: the thing (T); the action (A); the relation (R); and the value (V). These are called the **slots** of the narrative. When they are filled with VARs this is clear. But the notion of slots become less clear as we nest narrative inside one another. Should the sub-slots of sub-narratives be counted in the same way as the four parts of the outermost narrative? For now, we punt on this question. The four slots are loosely defined as noun, verb, relation, and noun or adjective. But they can be used any way you like. Several standard relations are

nar = event(thing, target, action) (example ”Bob kicked the ball”)

nar = attribute( thing, modifier) (example ”the ball is red”)

nar = sequence( first, second) (example ”we at cookies then we at pie”)

nar = cause( src, target ) (example: ”gunshots kill”)

nar = relation( thing, otherthing, relation, modifier) (example “The margins should be 2 mm below the gingiva”)

**NAR matching** and the goodness of fit score (**gof**). At the simplest level there are four slots in a NAR and we hope to match some of them to input text, one slot at a time. The goodness of fit is then evaluated in terms of whether all expected slots are filled, and whether the matched tokens in the text are closely or loosely spaced. We use a formula like this:

gof = (u/n) \*( r/f)

* u = num used slots of narrative
* n = num slots of narrative
* r = num words read (corrected for control words, dull words, and anything else I can skip)
* f = (last word read index) - (first word read index) + 1

[This is not the exact formula, there is a correction for when u==1, to enforce a score <=0.5 in that case.]

**Implicit slots.** In order to get more mileage out of NARs, arguments to the attribute(), event(), sequence(), cause(), and relation() can be specified inside of square brackets to indicate they are implicit. This means if the slot is filled the gof score uses it; if not, then both numerator and denominator of ”u/n” are decreased by one and the score is given as if that slot of the narrative was not required.

**Polarity** of a NAR. In some forms of language a sentence conveys a positive or negative sentiment. Typically “I was happy” versus “I was sad”. Like the true/false alternative of boolean logic, sentiment follows rules of combination such that the sentiment of the whole narrative can be analyzed in terms of the sentiment of the sub-narratives. Yet the rules are not boolean and are only imperfectly understood (See “AD HOC” little algorithms.] In order to support sentiment, a NAR has a field called polarity which is set to True (for “good”) by default. The field is also used when a NAR is negated.

In many cases the T, A, R, and V defining a NAR are themselves VARs. However the construction supports setting any of them to other NARs. For example, a causal relation could have another NAR as its target (eg a an adjective relation being the result of a causal relation). Thus NARs are, in principle, nestable to arbitrary depths. Such arbitrary depth NARs are supported completely in NWNReader.readText() however nesting is not supported in recordSlotEvents(). You will see the NWTopicReader using both types of reading. FYI, there is a “golden algorithm” that replaces nested reading as it is currently done, with a multi-level version of readSlotEvents() – I leave this to the future.

A NAR keeps a copy of the VARs and other NARs that define it. When a NAR is used for reading, its results, are stored internally, and can be extracted using TARV functions. You need to look at code to see examples. Mainly TARV gets more and more complicated as the NAR gets more nested. Yet I believe it works in a tidy way.

We say that a VAR is a NAR **of order 0**. When a NAR is built from VARs it is **of order 1**. Generally the order of a NAR is one greater than the order of any of its T, A, R, or V.

**Segmented Text** is probably one of the most important data types yet, since it is implemented as a simple list of VARs, it is nowhere defined in the code. Nevertheless segmented text represents a significant aspect of the implementation. In the old days reading text followed a flow like this:

text 🡪 tokenized text🡪 each token matched to each VAR in each NAR 🡪 final results

The significant improvement was to first convert the sequence of tokens into a sequence of VARs that match the tokens – called segmented text. It follows a flow like this:

text🡪tokenized text🡪segmented text (sequence of VARs)🡪 VAR matched to NAR 🡪 final result

This is very efficient, because the tokens are only read once, rather than once per NAR.

nwfind.py

Implements matching of tokens to VARs in the findInText(klist, tokens, rawtokens, itok, ifound) method.

Here the klist if that of a VAR, the tokens and rawtokens represent the input (“raw” means case is preserved), itok is an index in the token sequence, ifound is a list of indices in tokens (including but not limited to itok) where the tokens were used to perform a match. So a multi-word entry in a klist can match several of the tokens and all their indices will be recorded in the ifound.

findInText() relies on two type of processing. Unknown vars are represented by fixed string patterns like “ \_\_d\_\_ “ or “\_\_sufx\_\_” and they cause findInText to retrieve a string littoral. Normal fixed value VARs go, instead, through matchTOK() where the rules for special characters are applied.

nwcontrol.py

This file contains a tree of “operators” which are special VARs related to parsing and control of the meaning structure. This is where things like “boolean” operations are mentioned (AND, OR, BUT, NOT) as well as punctuations (!, comma,…). You will see that the **applyControl**() gives the ‘read’ an opportunity to pause and decide it a meaning has occurred and whether it should be archived (in the vault), or negated and archived, or etc.

nwsegment.py

As mentioned, the segmented text is an important (but not formally defined) data type. This file contains the PrepareSegment() code for converting tokens to VARs. It is also where the deepest “inner loop” reading occurs. In particular the ReadSegment() is a recursive function that will match text to a NAR, of arbitrary nesting depth.

ReadSegment() is one of two reading methods. The other recordSlotEvents() only works with NARs of order 1 or 0, that contain only VARs but not proper sub-NARs.

nwnreader.py

This file defines **NWNReader** class, which is the “last man standing” after a year or so of re-factoring things. They finally settled down to this class as the main APP level object to perform reading. The NWReader contains a NarVault, as described below.

nwvault.py

In principle, a NAR can be used to match text of arbitrary length. Every time a match occurs, the local match information is saved in a **NarVault**, so that when a long read is complete, all the information will still be accessible.

The code contains utilities for moving matched NAR information into the vaultduringanoperatation called **vaulting.** Vaulting proceeds in two steps: the information is staged in a “pre” vault. Then is either abandoned, modified, or moved to the more permanent part of the vault. These activities are contained in a variety of “rollUp” methods. Typically, what is already in “pre” is moved into the permanent vault to make room for something new being placed in pre. It gets complicated when a negation occurs that affects the previous NAR. By design, that NAR is still in “pre” where it can have its polarity reversed before it is moved to the permantent vault.

The nwvault.py file defines NarVault and includes GOF scoring, as well as the **recordSlotEvents**() method, which represents a different way of tracking nar match events on the way through a piece of text that may have multiple match events. The recordSlotEvents() is crucial for handling sentences like: “please make the m/d margins 2 mm subgingival and the b/l margins 3 mm”

To give a flavor of what goes on: the “and” is a control word defined by AND\_OP and causes a vaulting event. The end of the sentence is another control event and cause another (and final) vaulting. So the NAR will have opportunities for matching before and after the “and”.

nwchat.py

A chatbot is defined by the Read/Write/GOF interface: Text is passed in to Read(), there is a goodness of fit score GOF(), and the result is written with Write.

This is the only “application level” code still in Narwhal. It defines the **NWTopicReader** (which wraps NWNReader) as something that both performs NWNReader “read” methods and applies recordSlotEvents(). This ropic reader is combined with an **NWResponder** which is a state machine owning string outputs per each state. The combination of NWTopicReader and NWResponder is put into an abstract base class **TChat**.

One important derived class is **NWDataChat**. This is a TChat containing a data field which is to be updated when the TChat **update()** method is called. The NWDataChat is a model for any chatbot that do works to collect information from a user.

TChat supports several ways to combine chatbots. Through the CombinedChat a ‘+’ is defined allowing addition of two chatbots. (For example the base and core of an abutment each can have its own chatbot and a composite formed by adding them together). Another way to combine chatbots is visible in the orderchat.py file – where a chatbot contains a **currentChat** memberthat can be set to self or to a subchat.

**Examples.** The code currently supports these specific TChats, in various other directories of the GitHub repo.

**AboutChat** – to handle polite banter and report the chat version

**ConfirmChat** – to handle confirming an input

**abutmentchat** – to handle simple abutment descriptions

**orderchat** – to report on order status

**dentalchat** – the beginning of a larger service for dental ordering and information

**nbchat** – a prototype simple order system. Takes the user from start, through tooth numbering, abutment material, scanner, and confirmation id.

nwcontext.py

This file supports reading input text that refers to previously mentioned entities, using words like “it”, “them”, “what is the difference”, “both”, etc. It works like this: text is converted to segmented text on each call to read (recall that segmented text is a sequence of VARs.) So we store a copy of the segmented text, in reverse order, and then append additional reversed segments, as they occur.

text1🡪segmented text1🡪 reverse order sequence of VARs #1

text2🡪segmented text2🡪 reverse order sequence of VARs #2 is appended to sequence #1

etc.

The whole sequence is stored in an intermediate data type called **NWTopic**, which is an array of NWTopicReaders and is used to intialize a TChat.

Context retrieval goes a bit like this: a special VAR (like DIFF or IT) is present in a NAR whose read GOF score is not very good. Since the special VAR has a contextFn, this can be called with the stored VAR sequence, to retrieve missing but expected VAR from the past rather than the current input. These are retrieved by the context function and inserted into the input, as a second pass of reading is attempted. See the details in NWTopic.read().

nwlog.py

A simple logger that can be inserted into a TChat and appended to during read, and appended to during a write. I did not put it into the baseclass because (1) there will be plenty of derived classes that will not want to be logging at the same time; and (2) many derived classes implement their own versions of Read() and Write() and would need their own version of logger code.