
Red means uncertain/major revision needed
Pink means it is a suggestion but maybe not necessary
Type is a reserved keyword in Python
Variable
- var_type: Class
- name: str
+ get_type(): Class
+ get_name(): str
«Enumeration» WeekDay
Monday: 0
Tuesday: 1
Wednesday: 2
Thursday: 3
Friday: 4
Saturday: 5
Sunday: 6
«Enumeration» QuestionStatus
unanswered: 0
answered: 1
snoozed: 2
cancelled: 3
+ variables
DatabaseWriter
- conn: Connection
- variables: Set[Variable]
+ get_variables() -> Iterable[Variable]
+ execute_sql(sql_command: str)
+ execute_query(sql_command: str): List[Tuple]
var_2
var
After firing, only becomes fireable again when new changes occur in the database that make the rule fire. Otherwise the same output would be generated indifferently.
Output Production Process
«Abstract» Command
- is_executed: bool = False
- db_writer: DatabaseWriter
+ execute()
+ get_is_executed(): bool
QueryCommand
- is_executed: bool = False
- database: DatabaseWriter
- execute_sql_code: str
- laset_result: List[Tuple] = None
+ execute(): List[Tuple]
+ get_last_result(): List[Tuple]
+ get_is_executed(): bool
query
«Abstract» Rule
- var: Variable
- query: QueryCommand
- fireable: bool = False
+ get_var(): Iterable[Variable]
+ check_fireable(): bool
+ set_fired()
+ report_rows_of_interest(): List[Tuple]
ComparisonRule
- var_2: Variable
- relation: Union[~=, >, >=]
RelativeChangeRule
- period: float
relative_magnitude: float
Fire when a Variable made a relati large change (> magnitude) w.r.t. the average value during the past [period] time.
rules
«Abstract» CompositeRule
- rules: List[Rule]
ThresholdRule
- threshold: float
ANDCompositeRule
ORCompositeRule
Data Gather Process
Only allow sets of questions to be done together? (Can always just use a single question in QuestionOccurence, but we could enforce this)
«Abstract» ReversibleCommand
- is_executed: bool = False
- db_writer: DatabaseWriter
+ execute()
+ undo()
+ get_is_executed(): bool
Do we need redo functionality? Seems too time-consuming for our prototype, not?
Question
- content: str
- variable: Variable
+ get_content(): str
+ get_variable(): Variable
+ questions
QuestionOccurence
- status: QuestionStatus
- questions: Question
- snooze_time: int = 0
- id: int
+ get_id(): int
+ get_status(): QuestionStatus
+ set_status(new_status: QuestionStatus)
+ snooze(minutes: int)
Status superfluous? Snooze needs revision?
InsertCommand
- is_executed: bool = False
- database: DatabaseWriter
- execute_sql_code: str
- revert_sql_code: str = None
+ execute()
+ undo()
+ get_is_executed(): bool
Answer
- id: str
- timestamp: float
- variable: Variable
- value: variable.var_type
+ get_id(): int
+ get_value(): Any
+ extract_command(db_writer: DatabaseWriter): InsertCommand
OutputInvoker
- rules: Iterable[Rule]
+ mainloop(pipe: multiprocessing.Pipe)
- find_invoked_rules(): Iterable[Rule]
- invoke_rules():
convert_rows_to_feature_vect(rows: List[Tuple]): Tensor
OutputTextGenerator
- state: Tensor
«Abstract» torch.nn.Module
+ get_named_parameters(): Iterable[Tuple]
+ get_parameters(): Iterable[Parameter]
+ forward(x: Tensor): Tensor
RecurringQuestionSet
- days: Set[WeekDay]
- hour: int
- minutes: int
- variables: Set[Variable]
+ get_days(): Set[WeekDay]
+ get_hour(): int
+ get_minutes(): int
+ get_variables(): Set[Variable]
+ is_due(current_time: float, previous_check: float): bool
+ generate_questions(formulator: Formulator): Set[QuestionOccurence]
QuestionSchedule
- active_occurences: Dict[int, QuestionOccurence]
- snoozed_occurences: Dict[int, QuestionOccurence]
- finished_occurences: Dict[int, QuestionOccurence]
- prev_refresh_timestamp: float
- timeout_time: float
- question_sets: Set[ReoccuringQuestionSet]
- formulator: Formulator
+ get_active_occurences(): Set[QuestionOccurence]
+ register_answered(id: int)
Formulator
+ formulate_question(variable: Variable): str
Questions are stores in a dict that maps the ID of an Occurence to the QuestionOccurence itself.
BackendQuestioner
- executed_commands: List[Command]
- undone_commands: List[Command]
- schedule: QuestionSchedule
- db_writer: DatabaseWriter
+ get_active_questions(): Set[Question]
+ mainloop(pipe: multiprocessing.Pipe)
- add_new_variable(var: Variable)
- store_answer(answer: Answer)
- register_incoming_answers(): Iterable[Answer]
- send_questions(questions: Set[Question])
We will probably need to use multiprocessing. Perhaps a 'Pipe' to send data between processes. Here it will be used to exchange Questions and Answers. The BackendQuestioner will be able to match a Question to its Answer via their ID.
For commands such as undo or adding a variable we can make new classes. Then the BackendQuestioner can just extract stuff from the pipe, and check the type of the object (answer, new var, undo) to figure out what needs to be done with it.
Pipe
Pipe
GUI Process
GUI