

TaskGen Part 2

A Task-Based Agentic Framework building on StrictJSON

<https://github.com/simbianai/taskgen>

John Tan Chong Min

Acknowledgements:

Part of this work is done with Simbian AI (<https://simbian.ai/>)



TaskGen Showcase!

@douwe

@CowBelleh

Features of TaskGen (Part 1)

- Splitting of Tasks into subtasks for bite-sized solutions for each subtask
- Single Agent with LLM Functions
- Single Agent with External Functions
- Meta Agent with Inner Agents as Functions

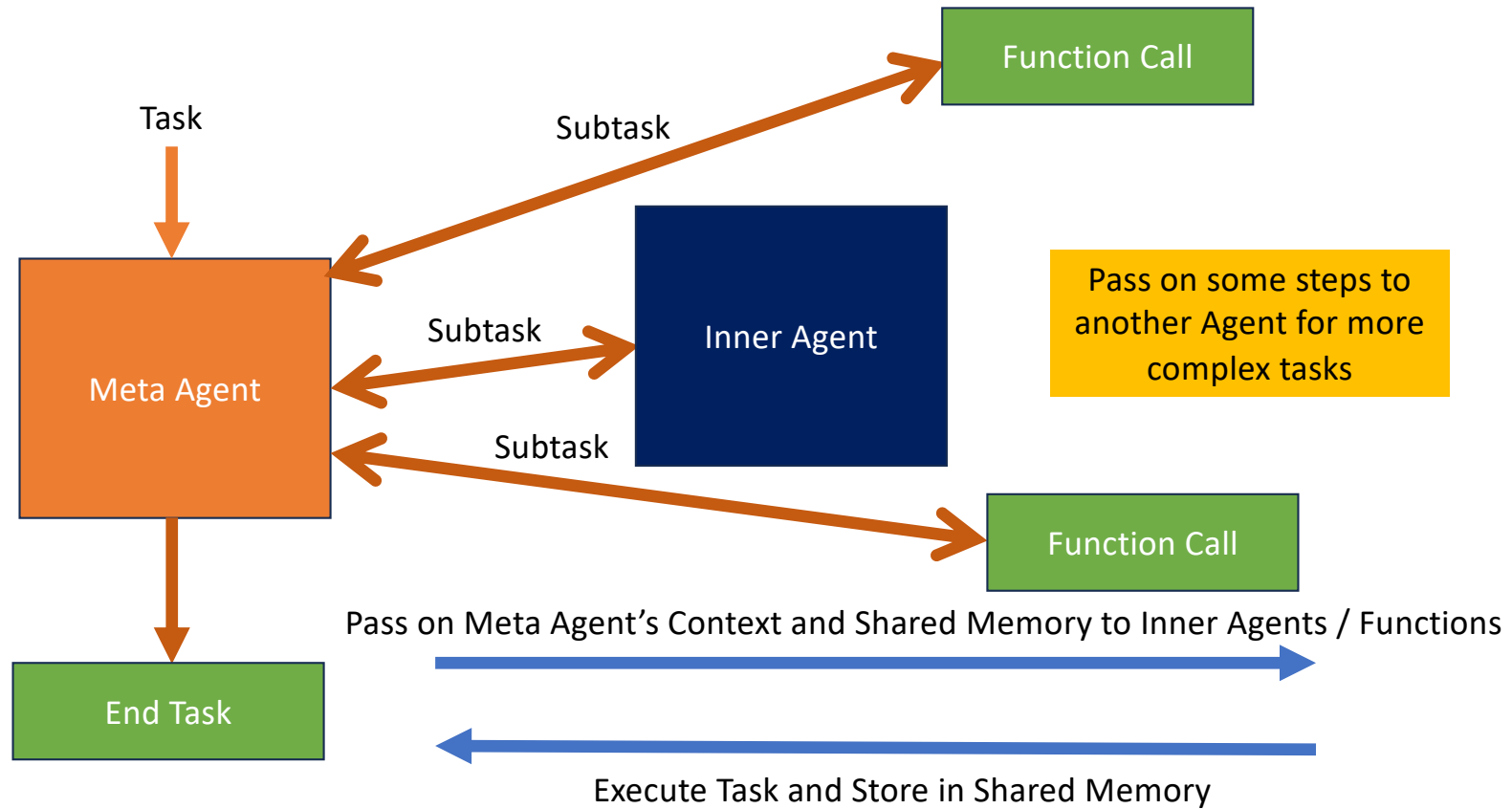
Features of TaskGen (Part 2 – This Session)

- **Shared Variables for multi-modality support**
- **Retrieval Augmented Generation (RAG) over Function space**
- **Memory to provide additional task-based prompts for task**
- **Additional Context for configuring your own prompts + add persistent variables**

TaskGen design Philosophies

- Instructions as concise as possible for minimal token use
- Hierarchical Agent Calling with no cycles
 - Ensure no infinite loop
 - Inner Agents the same structure as the Meta Agent for recursive implementation
 - Each level of agents get representation suitable for that level
- Generate a plan for fulfilling a task, and check plan at each step
 - Each step of the plan mapped to one function exactly to constrain output
 - Rule-based check to exit agentic loop once all parts of plan fulfilled

Recap: TaskGen



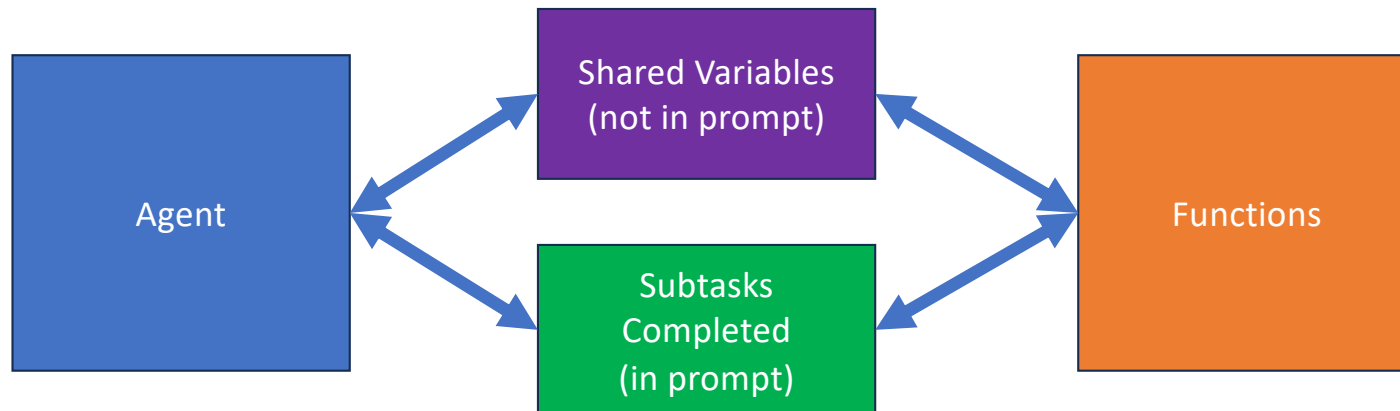
Shared Variables

Subtasks Completed

- Subtask: Find 5 words rhyming with 'pool'
 - pool, rule, fool, tool, school
- Subtask: Compose a 4-sentence poem using the words 'pool', 'rule', 'fool', 'tool', and 'school'
 - In the school, the golden rule is to never be a fool. Use your mind as a tool, and always follow the pool.

Shared Variables

- Sometimes you do not want to put everything into subtasks completed to reduce context length
- Useful for non-text modalities (e.g. audio, pdfs, image) and lengthy text modalities, which we do not want to output into subtasks completed directly



Shared Variables are accessible and modifiable by all functions!

LLM-Based Functions

```
# Function takes in increment (LLM generated) and s_total (retrieves from shared variable dict), and outputs to s_total (in shared variable dict)
add = Function(fn_description = "Add <increment: int> to <s_total>",
              output_format = {"s_total": "Modified total"})
```

External Functions

```
# Use shared_variables as input to your external function to access and modify the shared variables
def generate_quotes(shared_variables, number_of_quotes: int, category: str):
    ''' Generates number_of_quotes quotes about category '''
    # Retrieve from shared variables
    my_quote_list = shared_variables['s_quote_list']

    ### Add your function code here ###

    # Store back to shared variables
    shared_variables['s_quote_list'] = my_quote_list
```

Memory

Memory

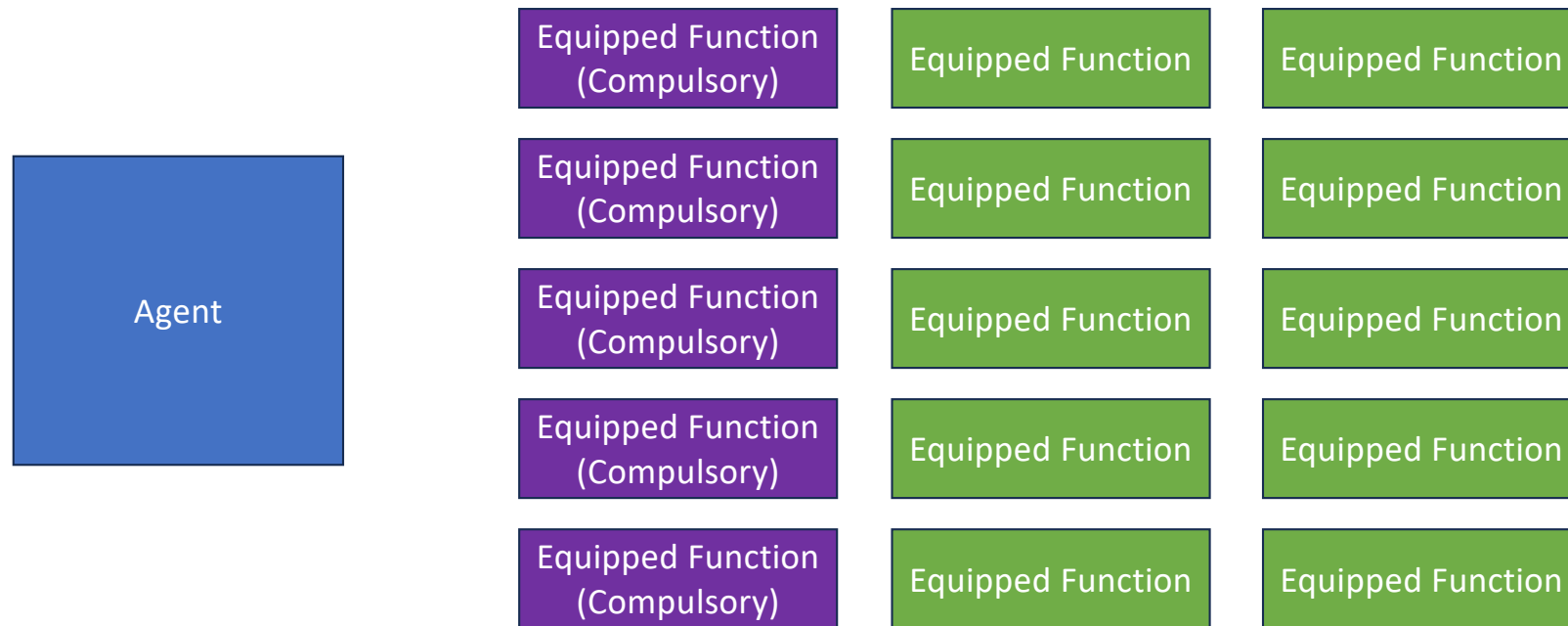
- **Problem:**

- Want to store lots of information from experiences / expert knowledge / past context but do not want to overload the agent's context

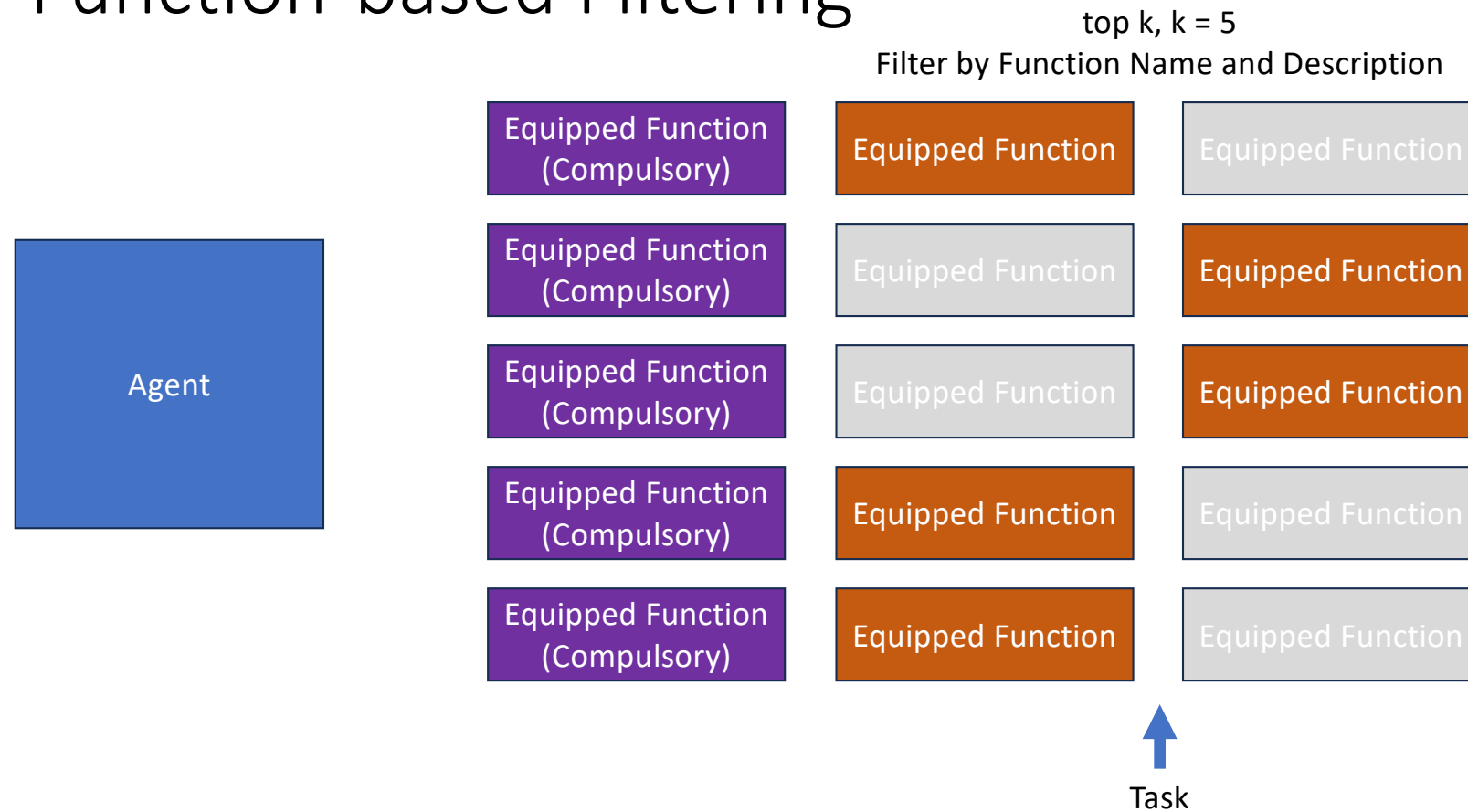
- **Solution:**

- Store **LOTS** of information in memory
- Only retrieve **LITTLE** based on the task at hand

Function-based Filtering



Function-based Filtering



Function-based Filtering (Example)

- Helps to reduce number of functions present in LLM context for more accurate generation

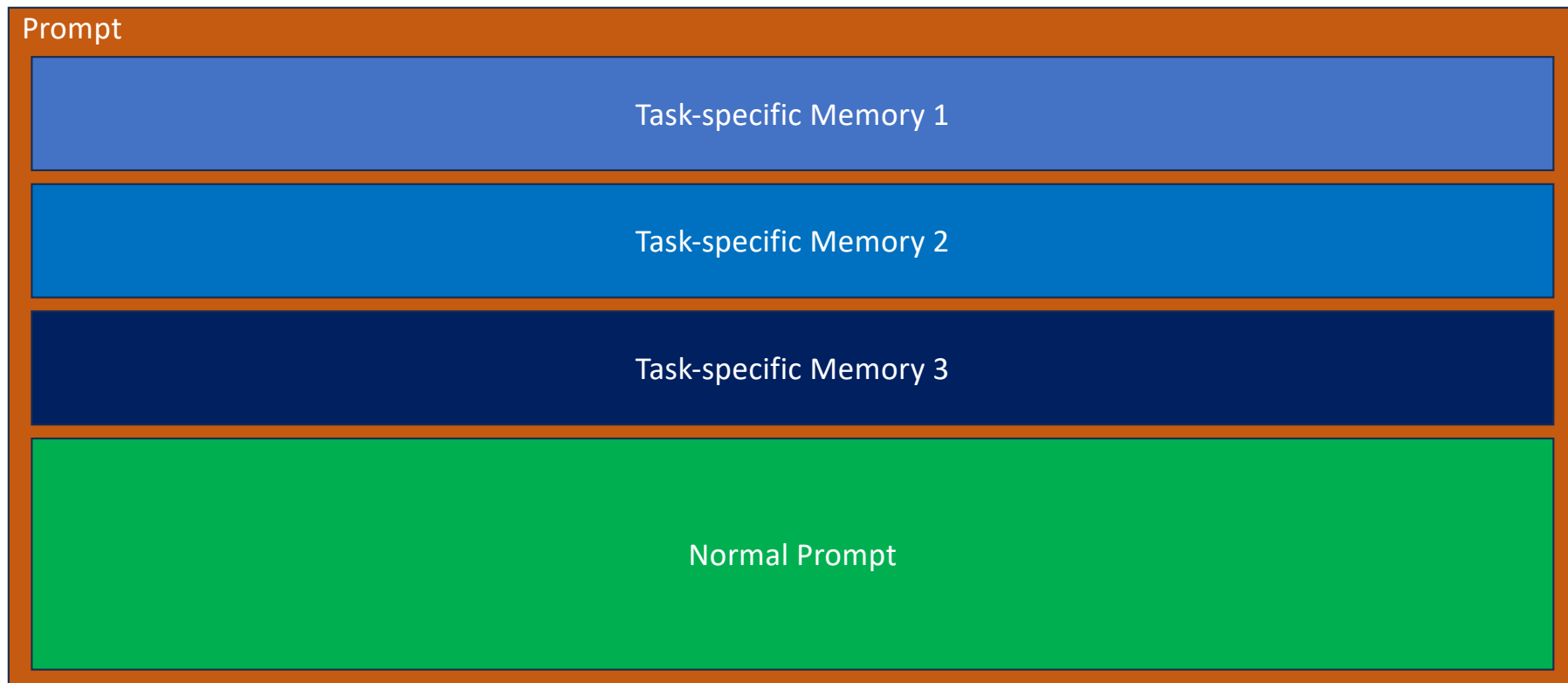
```
output = my_agent.run('Calculate 2**10 * (5 + 1) / 10')
```

Original Function List: add_numbers, subtract_numbers, add_three_numbers, multiply_numbers, divide_numbers, power_of, GCD_of_two_numbers, modulo_of_numbers, absolute_difference, generate_poem_with_numbers, List_related_words, generate_quote

Filtered Function Names: add_three_numbers, multiply_numbers, divide_numbers, power_of, modulo_of_numbers

Adding Task-based Instruction to Prompt

- Prompt can be modified according to Task



Examples

Adding Unknown Words to Meaning Mappings

```
my_agent.memory_bank['Number Meanings'] = Memory([{'Azo': 1}, {'Boneti': 2}, {'Andkh': 3}, {'Bdakf': 4}, {'dafdsk': 5},
{'ldsfn': 6}, {'sdkfn': 7}, {'eri': 8}, {'knewro': 9}, {'mdsnfk': 10}], # some nonsense words
top_k = 5, # choose top 5
mapper = lambda x: list(x.keys())) # we compare with the task using only the first word, e.g. Azo, Boneti, Andkh
```

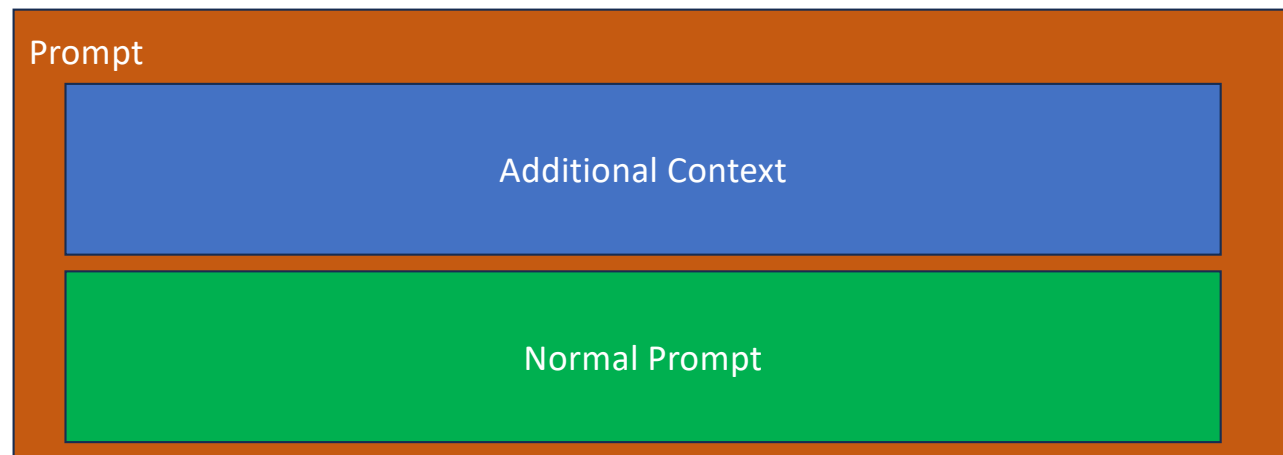
Adding Task to Function Mappings

```
my_agent.memory_bank['Sample Task to Function'] = Memory([
{'Sample Task': 'Find out about Azo + eri', 'Function': 'convert_word_to_number', 'list_of_words': ['Azo', 'eri']},
{'Sample Task': 'Evaluate 5 + 2', 'Function': 'sum_numbers_in_list', 'num_list': [5, 2]},
{'Sample Task': 'Find out about Boneti and Andkh', 'Function': 'convert_word_to_number', 'list_of_words': ['Boneti', 'Andkh']},
{'Sample Task': 'What is Andkh?', 'Function': 'convert_word_to_number', 'list_of_words': ['Andkh']},
{'Sample Task': 'Booyah', 'Function': 'generate_quote', 'topic': 'TaskGen'}
],
top_k = 3, # choose top 3
mapper = lambda x: x['Sample Task']) # we compare with the task using only with the user query
```

Additional Context

Additional Context

- **Additional Context** for Agents to store persistent variables, e.g. Agent state like ingredients remaining, money remaining, position in maze
- Can also store conversation between Agent and User and use it as context for inferring what User wants



Additional Context (Example)

- With Additional Context, we can actually reset Subtasks Completed every task because important information across tasks is all stored here

```
def get_additional_context(agent):  
    ''' Outputs additional information to the agent '''  
  
    # process additional context based on shared variables  
    # (this is what is called persistent variables - variables that will be updated each step)  
    additional_context = f'''User Money Remaining: ``{agent.shared_variables["money_remaining"]}``  
Items Searched: ``{agent.shared_variables["items_searched"]}``  
Items Purchased: ``{agent.shared_variables["purchased_items"]}``  
Past Conversation: ``{agent.shared_variables["conversation"]}``'''  
  
    return additional_context  
  
# assign this to agent's additional context  
my_agent.get_additional_context = get_additional_context
```

New Feature: Saving and Loading Agents

Example 1: Saving Agent

```
my_agent.save_agent('myagent.pkl')
```



Example Output

```
Agent saved to myagent.pkl
```

Example 2: Loading Agent

```
new_agent = Agent().load_agent('myagent.pkl')
```



Example Output

```
Agent loaded from myagent.pkl
```

Questions to Ponder

- How can memory be stored in multiple abstraction spaces and how do we choose the abstraction spaces to use?
- What could be a better way to retrieve memory than cosine similarity?
- How can Agent infer the task if it is not provided by user directly, e.g. conversational interface?

Part 1 Slides

StrictJSON



A JSON parser for preserving all keys in JSON with type checking and more


<https://github.com/tanchongmin/strictjson>


Why StrictJSON?


- JSON is a much less verbose method of output for LLM


Designing: Programming Language


: CEO (Chief Executive Officer) : CTO (Chief Technology Officer)

: To satisfy the new user's demand and make a desktop application realizable, you should propose a concrete programming language.

: Python with the Pygame library is a suitable choice for implementing a basic Gomoku game.

: Great! Python is a versatile and widely-used programming language that will be suitable for implementing a basic Gomoku game. Its simplicity and readability make it a popular choice for beginners and experienced developers alike.

: Let's get started with the development of the basic Gomoku game in Python using the Pygame library!

: <END> Python.

ChatDev – Natural Text

⚡ Default (GPT-3.5)

C

I want to create a Gomoku game.

Output the following json without explanation

{'programming language': 'programming language to create it',
'modules': 'corresponding modules to import'}



```
{  
  "programming language": "Python",  
  "modules": "pygame"  
}
```

JSON

My Critique on “Communicative Agents for Software Development. Qian et al. 2023.”

Why StrictJSON?

- Much lower token usage

```
{
  "type": "function",
  "function": {
    "name": "get_current_weather",
    "description": "Get the current weather",
    "parameters": {
      "type": "object",
      "properties": {
        "location": {
          "type": "string",
          "description": "The city and state, e.g. San Francisco, CA",
        },
        "format": {
          "type": "string",
          "enum": ["celsius", "fahrenheit"],
          "description": "The temperature unit to use. Infer this from the users location.",
        },
      },
      "required": ["location", "format"],
    },
  },
}
```

OpenAI Function Calling – Just Defining Function

```
strict_json(system_prompt = 'Fit user intent into a function call',
             user_prompt = 'Get current weather in San Francisco',
             output_format = {'Location': 'City and state, type: str',
                              'Temperature': 'Infer from location, type: Enum["celsius","fahrenheit"]'})

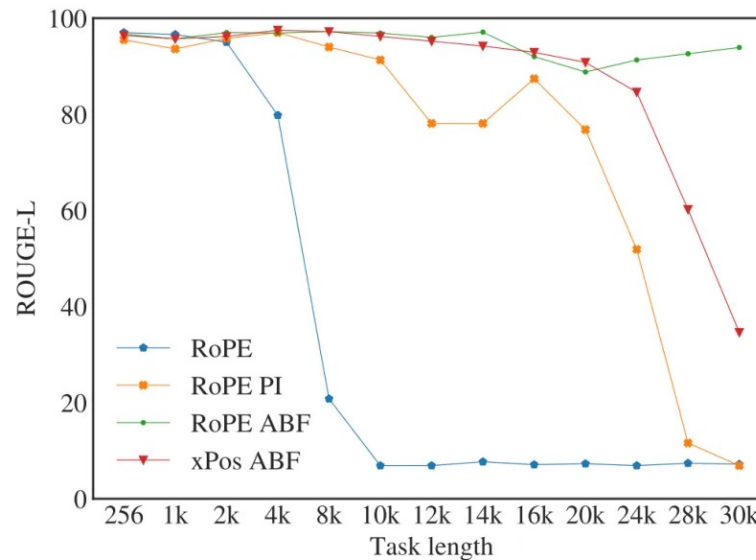
{'Location': 'San Francisco, CA', 'Temperature': 'celsius'}
```

StrictJSON – Defining and getting function params

<https://platform.openai.com/docs/guides/function-calling>

Tokens impact not just cost, but performance

- Performance sharply degrades after 2-3k tokens
 - For Rotary Positional Embeddings (RoPE) in Llama 2



(b) Performance on FIRST-SENTENCE-RETRIEVAL task.

Effective Long-Context Scaling of Foundation Models. 2023. Xiong et. al.

StrictJSON Ensures Keys and Types of Output

- Currently supported types:
 - `int`, `float`, `str`, `dict`, `list`, `array`, `Dict[]`, `List[]`, `Array[]`, `Enum[]`, `bool`

Example Usage 1

```
res = strict_json(system_prompt = 'You are a classifier',
                  user_prompt = 'It is a beautiful and sunny day',
                  output_format = {'Sentiment': 'Type of Sentiment, type: Enum["Pos", "Neg", "Other"]',
                                   'Adjectives': 'Array of adjectives, type: List[str]',
                                   'Words': 'Number of words, type: int',
                                   'In English': 'Whether sentence is in English, type: bool'})

print(res)
```

Example Output 1

```
{'Sentiment': 'Pos', 'Adjectives': ['beautiful', 'sunny'], 'Words': 7, 'In English': True}
```

StrictJSON is able to output complex code

Example Usage

```
res = strict_json(system_prompt = 'You are a code generator, generating code to fulfil a task',
                  user_prompt = 'Given array p, output a function named func_sum to return its sum',
                  output_format = {'Elaboration': 'How you would do it',
                                   'C': 'Code',
                                   'Python': 'Code'})

print(res)
```

Example Output

```
{'Elaboration': 'Use a loop to iterate through each element in the array and add it to a running total.',
```

```
'C': 'int func_sum(int p[], int size) {\n    int sum = 0;\n    for (int i = 0; i < size; i++) {\n        sum += p[i];\n    }\n    return sum;\n}',
```

```
'Python': 'def func_sum(p):\n    sum = 0\n    for num in p:\n        sum += num\n    return sum'}
```

StrictJSON supports nested structure!

Example Input

```
res = strict_json(system_prompt = 'You are a classifier',
                  user_prompt = 'It is a beautiful and sunny day',
                  output_format = {'Sentiment': ['Type of Sentiment',
                                                  'Strength of Sentiment, type: Enum[1, 2, 3, 4, 5]'],
                                  'Adjectives': "Name and Description, type: List[Dict['Name', 'Description']]",
                                  'Words': {
                                      'Number of words': 'Word count',
                                      'Language': {
                                          'English': 'Whether it is English, type: bool',
                                          'Chinese': 'Whether it is Chinese, type: bool'
                                      },
                                      'Proper Words': 'Whether the words are proper in the native language, type: bool'
                                  }
                  })

print(res)
```

Example Output

```
{
  'Sentiment': ['Positive', 3],
  'Adjectives': [{
    'Name': 'beautiful',
    'Description': 'pleasing to the senses'
  }, {
    'Name': 'sunny',
    'Description': 'filled with sunshine'
  }],
  'Words': {
    'Number of words': 6,
    'Language': {
      'English': True,
      'Chinese': False
    },
    'Proper Words': True
  }
}
```

How StrictJSON works

- Uses ### delimiters to enclose keys, then extract them via regex

```
strict_json(system_prompt = 'Fit user intent into a function call',  
            user_prompt = 'Get current weather in San Francisco',  
            output_format = {'Location': 'City and state, type: str',  
                             'Temperature': 'Infer from location, type: Enum["celsius","faranheit"]'},  
            verbose = True)
```

System prompt: Fit user intent into a function call

Output in the following json string format: {'###Location###': '<City and state, type: str>', '###Temperature###': '<Infer from location, type: Enum["celsius","faranheit"]>'}

Update text enclosed in <>. Be concise. Output only the json string without any explanation.

User prompt: Get current weather in San Francisco

GPT response: {'###Location###': 'San Francisco, California', '###Temperature###': 'celsius'}

{'Location': 'San Francisco, California', 'Temperature': 'celsius'}

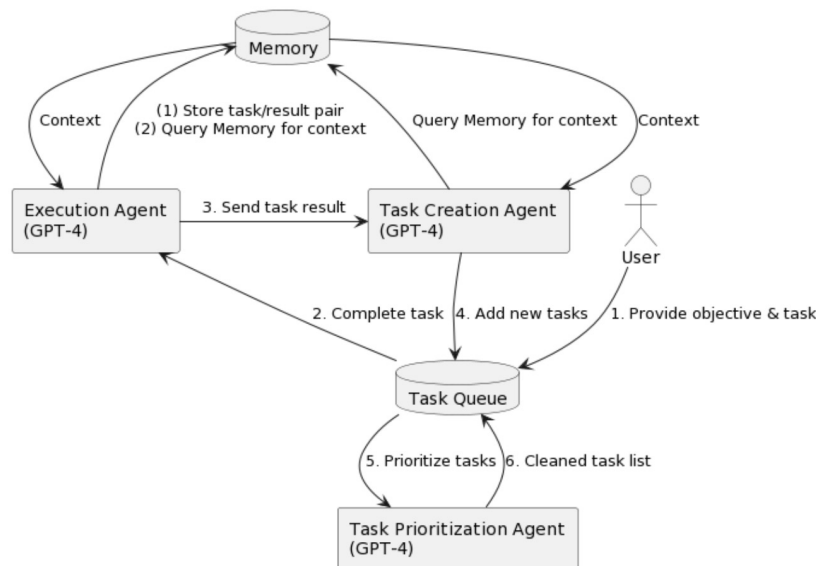
TaskGen

Building on StrictJSON for Agentic Task Solving!

<https://github.com/simbianai/taskgen>

Problems with existing Agentic Framework

- Never-ending loop of tasks



Yohei Nakajima's BabyAGI
Related - AutoGPT

- Too verbose
- Agents respond to tasks they are not meant to

Coder (to chat_manager):

Thank you for understanding. If you have access to the full paper, I recommend reading it thoroughly to get a comprehensive understanding of the research methodology, results, and discussions. This will allow you to evaluate the strengths and weaknesses in more detail and within the context of the AutoGen field. Additionally, you can consider factors such as the paper's impact on the field, the clarity of presentation, the quality of references, and any limitations or future directions stated by the authors. This holistic assessment will provide a more accurate and nuanced evaluation of the paper.

Engineer (to chat_manager):

You're welcome! I'm glad I could help. If you have any more questions or need further assistance, feel free to ask. Happy reading and evaluating the paper!

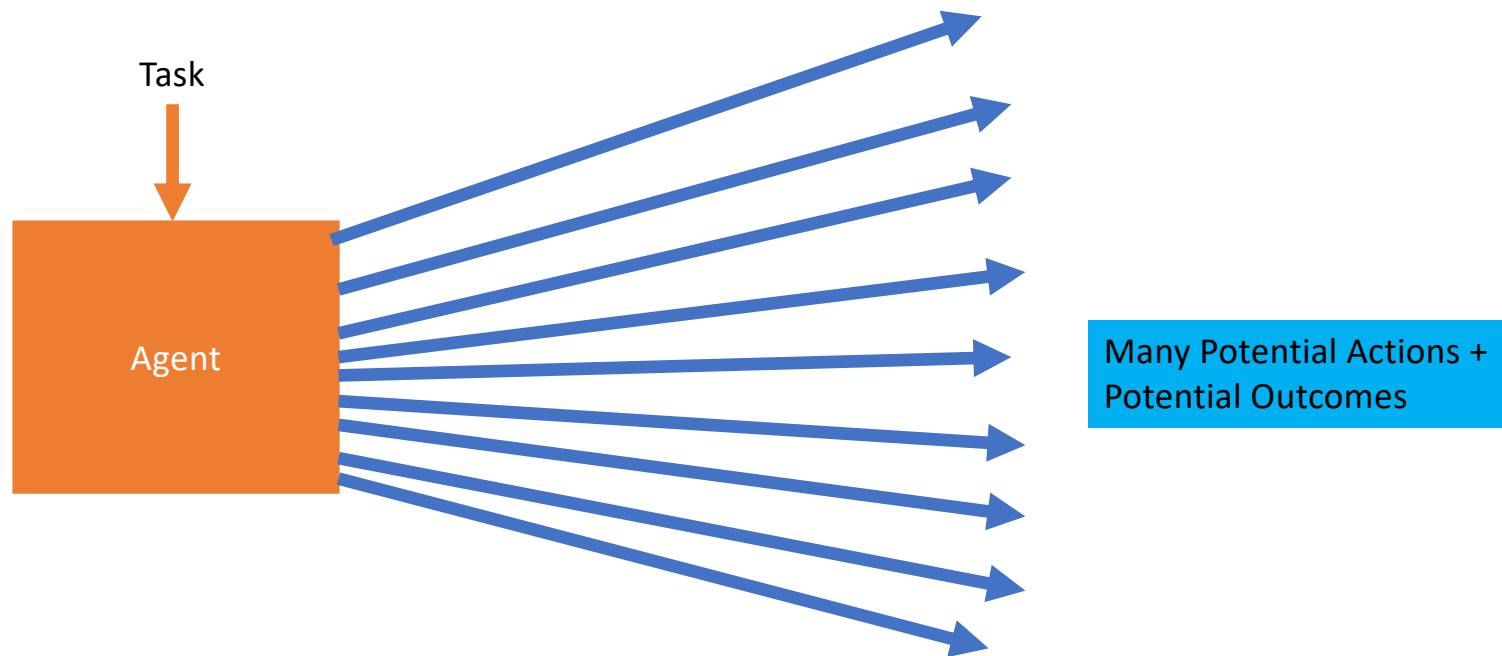
>>>>>> USING AUTO REPLY...
User_proxy (to chat_manager):

Coder (to chat_manager):

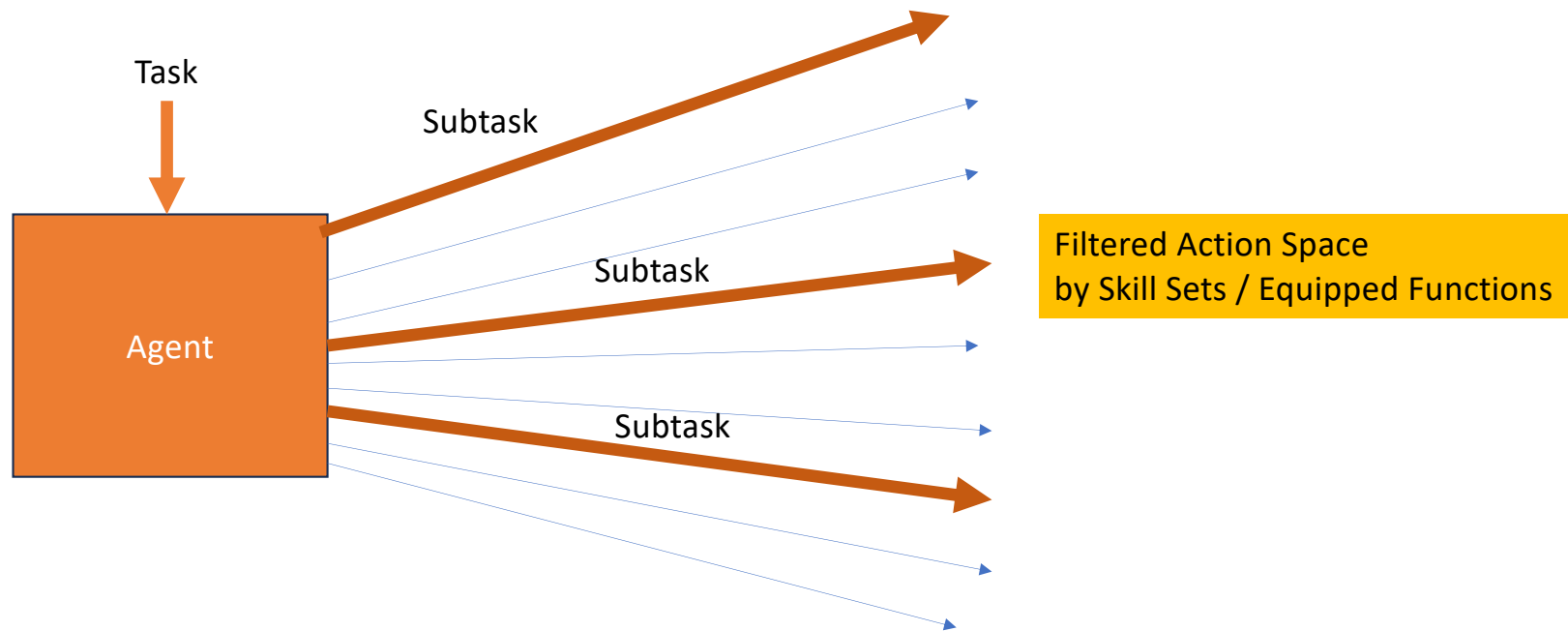
Thank you! I appreciate your assistance. If I have any further questions or need more guidance, I will reach out to you. Have a great day!

AutoGen
Related – crew.ai

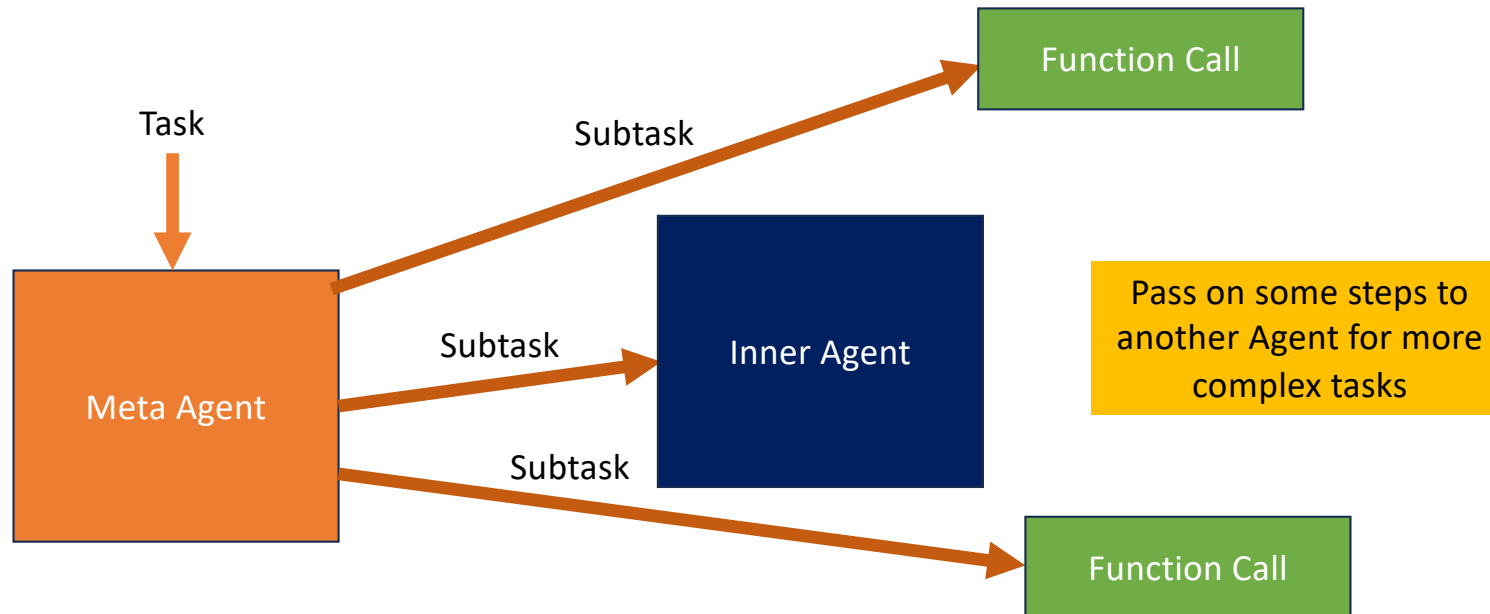
Key Question: How does an agent know what to do?



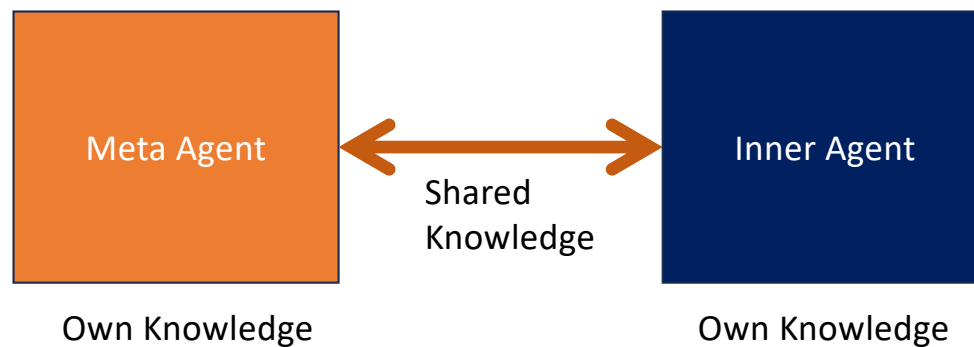
Key Question: How does an agent know what to do?



Key Question: How does an agent know what to do?



Key Question: How much to know?



Subtasks Completed

{“Think of 5 dishes”: “Pasta,
pizza, truffle, tiramisu, lasagna”,
“Come up with pricing”: [“\$3”,
“\$4”, “\$5”, “\$8”, “\$11”]}

Agent Initialisation

```
# Create your agent by specifying name and description  
my_agent = Agent('Helpful assistant', 'You are a generalist agent')
```

```
# Show the agent status - By default agent comes equipped with default function `use_llm` which queries the llm  
# end_task is to end the current task if it is completed  
my_agent.status()
```

```
Agent Name: Helpful assistant  
Agent Description: You are a generalist agent  
Available Functions: ['use_llm', 'end_task']  
Task: No task assigned  
Subtasks Completed: None  
Is Task Completed: False
```

Agent Task Running

```
# Do the task by subtasks. This does generation to fulfil task  
output = my_agent.run('Give me 5 words rhyming with cool, and make a 4-sentence poem using them')
```

Subtask identified: Find 5 words rhyming with cool

Getting LLM to perform the following task: Find 5 words rhyming with cool

> pool, drool, fool, school, tool

Subtask identified: Create a 4-sentence poem using the rhyming words

Getting LLM to perform the following task: Create a 4-sentence poem using the rhyming words

> In the school by the pool, a fool used a tool, making others drool.

Task completed successfully!

Agent Reply and Status

```
# Generates a meaningful reply to the user about the task according to its current state. Functions like a QA bot  
output = my_agent.reply_user()
```

In the school by the pool, a fool used a tool, making others drool. Words rhyming with cool: pool, drool, fool, school, tool.

```
# see the updated agent status  
my_agent.status()
```

Agent Name: Helpful assistant

Agent Description: You are a generalist agent

Available Functions: ['use_llm', 'end_task']

Task: Give me 5 words rhyming with cool, and make a 4-sentence poem using them

Subtasks Completed:

Subtask: Find 5 words rhyming with cool

pool, drool, fool, school, tool

Subtask: Create a 4-sentence poem using the rhyming words

In the school by the pool, a fool used a tool, making others drool.

Subtask: Give me 5 words rhyming with cool, and make a 4-sentence poem using them

In the school by the pool, a fool used a tool, making others drool. Words rhyming with cool: pool, drool, fool, school, tool.

Is Task Completed: True

Agent Functions

Example Internal Function

```
sentence_style = Function(fn_description = 'Output a sentence with <obj> and <entity> in the style of <emotion>',  
                          output_format = {'output': 'sentence'})
```

Example External Function

```
def binary_to_decimal(x):  
    return int(str(x), 2)
```

a single-valued external function, with more expressive variable description

```
b2d = Function(fn_description = 'Convert input <x>: a binary number in base 2> to base 10',  
              output_format = {'output1': 'x in base 10'},  
              external_fn = binary_to_decimal)
```

Docstring must provide all compulsory input variables

*# We will ignore shared_variables, *args and **kwargs*

```
def add_number_to_list(num1: int, num_list: list, other_var: bool = True, *args, **kwargs):  
    '''Adds num1 to num_list'''  
    num_list.append(num1)  
    return num_list
```

```
fn = Function(external_fn = add_number_to_list,  
             output_format = {'num_array': 'Array of numbers'})
```

```
str(fn)
```

```
"Description: Adds <num1: int> to <num_list: list>\nInput: ['num1', 'num_list']\nOutput: {'num_array': 'Array of numbers'}\n"
```

Newest feature:

Automatic
function
representation
by parsing
docstring

Concise Representation of Functions

```
# Create your agent
my_agent = Agent(agent_name = 'Helpful assistant',
                  agent_description = 'You are a generalist agent')
```

```
# Assign functions
my_agent.assign_functions(function_list = [b2d, sentence_style])
```

```
<taskgen.agent.Agent at 0x10b735510>
```

```
# Show the functions the agent has
my_agent.print_functions()
```

```
Name: use_llm
Description: Used only when no other function can do the task
Input: []
Output: {'Output': 'Output of LLM'}
```

```
Name: end_task
Description: Use only when task is completed
Input: []
Output: {}
```

```
Name: binary_to_decimal
Description: Convert input <x: a binary number in base 2> to base 10
Input: ['x']
Output: {'output1': 'x in base 10'}
```

```
Name: generate_sentence_with_emotion
Description: Output a sentence with <obj> and <entity> in the style of <emotion>
Input: ['obj', 'entity', 'emotion']
Output: {'output': 'sentence'}
```

<< 50% of the
equivalent
tokens for
OpenAI Function
Calling

Multi-step usage of Functions

```
[11]: # multi-task with multiple functions
      output = my_agent.run('Generate me a happy sentence with a number (convert 1001 to decimal) and a ball')

      Subtask identified: Convert the binary number 1001 to decimal
      Calling function binary_to_decimal with parameters {'x': 1001}
      > {'output1': 9}

      Subtask identified: Generate a happy sentence with the converted number and a ball
      Calling function generate_sentence_with_emotion with parameters {'obj': 'the number 9', 'entity': 'a ball', 'emotion': 'happiness'}
      > {'output': 'The number 9 brings me so much happiness, just like a ball does.'}

      Task completed successfully!

[12]: # visualise the subtask outputs of the task
      print(output)

      [{'output1': 9}, {'output': 'The number 9 brings me so much happiness, just like a ball does.'}]

[13]: # give a response to user
      output = my_agent.reply_user()

      The number 9 brings me so much happiness, just like a ball does.
```

Alternative to OpenAI Function Calling

```
# this should call generate_sentence_with_emotion
my_agent.reset()
function_name, function_params = my_agent.select_function(
    task = 'Output a sentence with bell, dog and happy')
print(f'Selected function name: {function_name}\nSelected function params: {function_params}')

my_agent.use_function(function_name, function_params, stateful = False)
```

```
Selected function name: generate_sentence_with_emotion
Selected function params: {'obj': 'bell', 'entity': 'dog', 'emotion': 'happy'}
Calling function generate_sentence_with_emotion with parameters {'obj': 'bell', 'entity': 'dog', 'emotion': 'happy'}
> {'output': "The dog's tail wagged happily at the sound of the bell."}
```

Under the Hood – What is done for Planner

- **"Thoughts"**: "How to do Assigned Task"
- **"Overall Plan"**: "Array of steps to complete Assigned Task from beginning to end, type: list"
- **"Reflection"**: "What has been done and what is still left to do"
- **"Overall Plan Completed"**: "Whether array elements in Overall Plan are already completed, type: List[bool]"
- **"Next Step"**: "First non-completed element in Overall Plan"
- **"Equipped Function"**: f"Name of Equipped Function to use for Next Step, type: Enum{list(self.function_map.keys())}"
- **"Instruction"**: "Instruction for the Equipped Function if any"

Key Ideas:

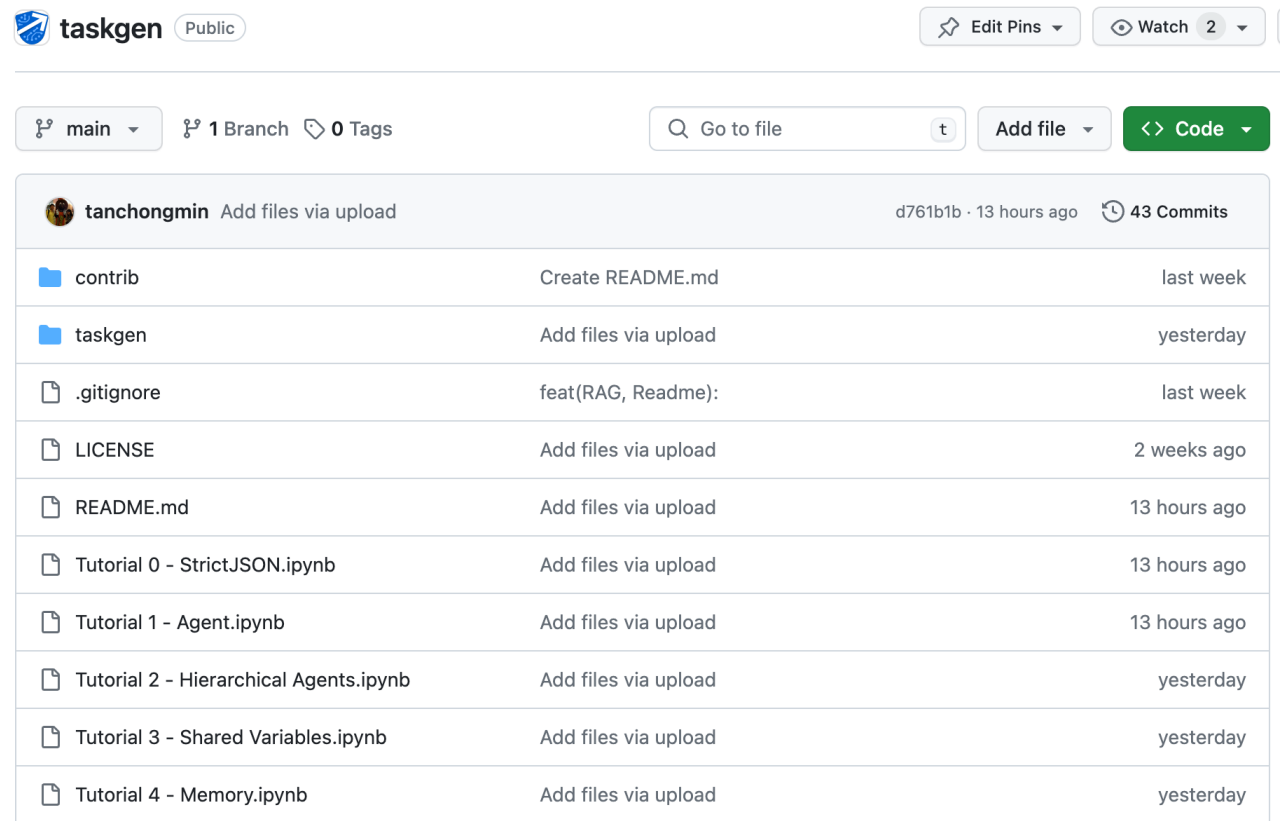
Chain of Thought,
Reflection,
List-based planning,
Constrained Generation,
Rule-based ending of task

Advanced Features

- **Shared Variables** help to store non-text modality and lengthy text for reference in any function
- **Memory** helps perform Retrieval Augmented Generation over Function space based on task
- **Memory** also helps augment additional information in input prompt based on task

How you can help

- TaskGen is meant to help us move towards better agentic structures
- Free to use, even commercially
- Help to like and star the GitHub
- Help to contribute example External Functions / Jupyter Notebooks for more boilerplate code for others!



The screenshot shows the GitHub repository page for 'taskgen' by 'tanchongmin'. The repository is public and has 1 branch and 0 tags. The main branch is selected. The repository contains several files and folders, including 'contrib', 'taskgen', '.gitignore', 'LICENSE', 'README.md', and four tutorials. The commit history shows that the repository was created 13 hours ago and has 43 commits.

File/Folder	Commit Message	Commit Time
contrib	Create README.md	last week
taskgen	Add files via upload	yesterday
.gitignore	feat(RAG, Readme):	last week
LICENSE	Add files via upload	2 weeks ago
README.md	Add files via upload	13 hours ago
Tutorial 0 - StrictJSON.ipynb	Add files via upload	13 hours ago
Tutorial 1 - Agent.ipynb	Add files via upload	13 hours ago
Tutorial 2 - Hierarchical Agents.ipynb	Add files via upload	yesterday
Tutorial 3 - Shared Variables.ipynb	Add files via upload	yesterday
Tutorial 4 - Memory.ipynb	Add files via upload	yesterday

<https://github.com/simbianai/taskgen>

Questions to Ponder

- Planner might sometimes generate a similar next step as before. How can we mitigate that?
- How can we integrate various forms of memory into an Agent?
- What level of context should each Agent/function should have access to for best performance?