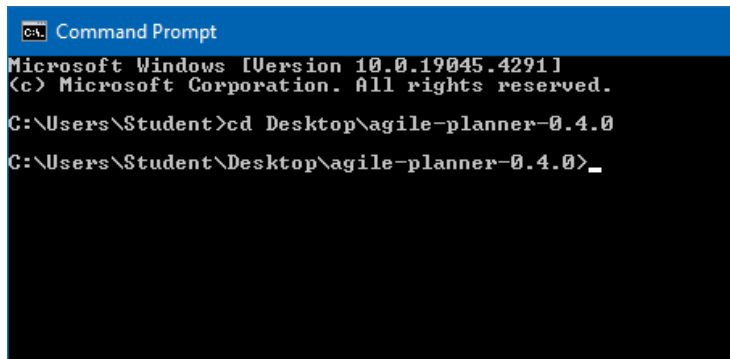


Agile Planner v0.4.0 [pre-release]

Getting started:

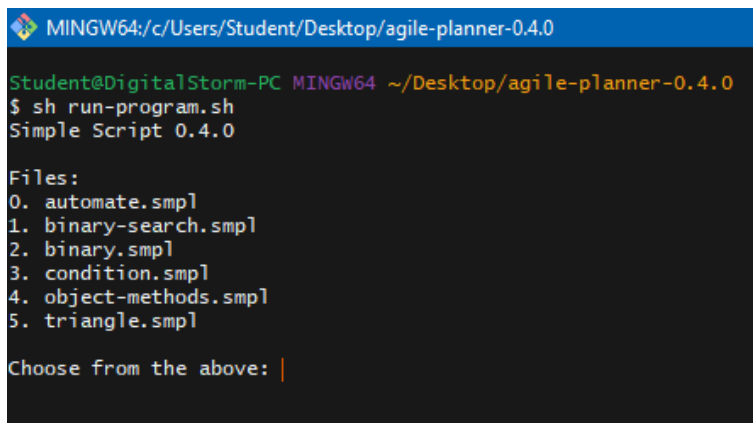
The current version of Agile Planner is CLI-based and as such requires a basic understanding of how to properly work with your terminal. In this guide, I'll be showing two primary options: DOS and Unix/Linux (Mac has the Unix/Linux option).

For Microsoft DOS, you'll need to change your base directory over to '**agile-planner-0.4.0**'. To do so, simply use the 'cd' command like the following:



```
Command Prompt
Microsoft Windows [Version 10.0.19045.4291]
(c) Microsoft Corporation. All rights reserved.
C:\Users\Student>cd Desktop\agile-planner-0.4.0
C:\Users\Student\Desktop\agile-planner-0.4.0>_
```

For Unix/Linux, you'll use the same 'cd' command to switch over to the agile planner folder as well like below:



```
MINGW64:/c/Users/Student/Desktop/agile-planner-0.4.0
Student@DigitalStorm-PC MINGW64 ~/Desktop/agile-planner-0.4.0
$ sh run-program.sh
Simple Script 0.4.0

Files:
0. automate.smp1
1. binary-search.smp1
2. binary.smp1
3. condition.smp1
4. object-methods.smp1
5. triangle.smp1

Choose from the above: |
```

The big difference between the two is how you display the contents of your folder (i.e. **ls** for Unix/Linux and **dir** for DOS) and how to clear your screen of text (i.e. **clear** for Unix/Linux and **cls** for DOS). With this basic understanding out of the way, we're ready to actually start the program!

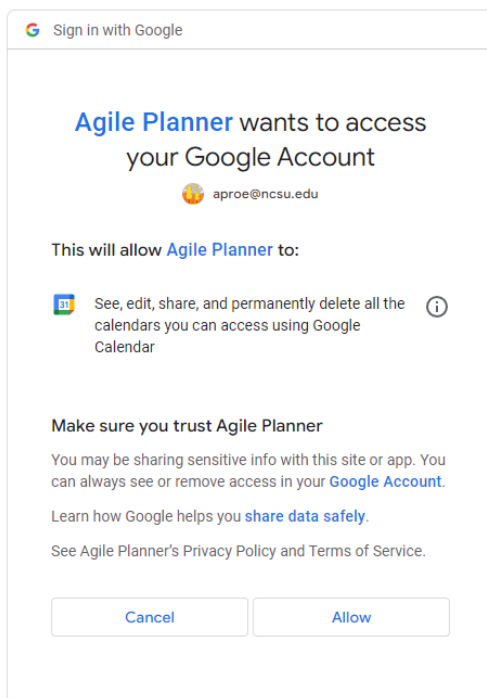
Running Agile Planner:

At this point, it doesn't matter which terminal you use as the commands going forward will be exactly the same. To start off the program, we will enter the following:

```
java -jar agile-planner-0.4.0-SNAPSHOT.jar
```

Once this has been entered, you will be prompted with the name of the program followed by a request to "Please open the following address in your browser: ..."

Once you head over to the webpage, which should have opened up automatically, you'll be greeted with a Google Authorization prompt. After choosing which account you'd like to work with, you'll then be asked to allow Agile Planner access to your calendar:



If you allow access, you'll then be able to continue on with the app and get started!

NOTE: We are currently in testing stage with this app and thus have limited users with access to the Google Calendar functions. If you are interested, contact us via aproe@ncsu.edu for access.

An Introduction to Simple Script

Simple is an Object-Oriented interpreted programming language that functions similar to Python. Its syntax can be summarized as the following:

```
#####
```

```
include: __CURR_CONFIG__, __LOG__, __HTML__
```

```
# Imports the scheduling data for the session
```

```
jbin_file: input_word("Import JBin -> ")
```

```
import_schedule(jbin_file)
```

```
# Creates a Card
```

```
func create_card()
```

```
    flag: input_bool("Create Card(T/F) -> ")
```

```
    if (flag)
```

```
        _name: input_line("Name -> ")
```

```
        _c1: card(_name)
```

```
        create_card()
```

```
    else
```

```
        println()
```

```
# Integrates new Cards with system and displays the Board
```

```
create_card()
```

```
add_all_cards()
```

```
display_board()
```

```
# Processes a Task by assigning it to the relevant Card
```

```
func process_card(_t1)
```

```
    idx: input_int("Card Index -> ")
```

```
    if (idx.<(0))
```

```
        return
```

```
    else
```

```
        c1: get_card(idx)
```

```
        c1.add(_t1)
```

```

# Creates a Task with user inputs and assigns a bool flag as a check
task_added: false
func create_task()
    flag: input_bool("Create Task(T/F) -> ")
    if (flag)
        task_added: true
        _name: input_line("Name -> ")
        _hours: input_int("Hours -> ")
        _days: input_int("Days -> ")
        _t1: task(_name, _hours, _days)
        process_card(_t1)
        create_task()
    else
        println()
        return

# Begins the process of creating a Task, displaying the Board, and rebuilding the schedule
create_task()
if (task_added)
    display_board()
    add_all_tasks()
    build()
    export_google()

# Displays the schedule
println("\n")
display_schedule()

# Exports the scheduling data to the user provided file
jbin_file: input_word("Export JBin -> ")
export_schedule(jbin_file)

#####

```

We will cover each of the core components along with a list of built-in functions and data types/methods.

Include Flags:

This part of the script tells Simple what to include when it's interpreting your code. This can range from type of configuration settings to logging out the stack trace. Here's all the possible options:

```
__CURR_CONFIG__ (Uses the current config settings)
__DEF_CONFIG__  (Uses the default config settings)
__HTML__        (Generates an HTML page for the session)
__LOG__         (Stores a log of the stack trace)
```

Variables:

Variables, similar to Python, are dynamic in nature and allow switching between types as often as you please. Declaration and instantiation are done simultaneously in Simple and cannot be separated. Below is the syntax:

```
<var_name>: <data_type>(<arg1>, <arg2>, ...>)
```

And the following are examples of how you could create an instance of each built-in type:

```
c1: card("Calc 3")
my_task: task("Math HW", 4, 2)
var: label("HW", 2)
_c1: checklist("ToDo")
x: 34
str: "Hello world"
flag: false
```

Whenever a variable is attempting to hold an object instance, it must attach the ':' to the end of its name. Thankfully, memory is dynamic here and allows for switching of types with storage like the following:

```
c1: card("Calc 3")
c1: "This is some string"
c1: 34
# Will print out the value of '34'
println(c1)
```

Built-In Functions:

Simple script provides an extensive list of built-in functions that solve a wide variety of problems. We will cover some of the more important ones and a link will be included to the entire list in [Appendix A](#).

These two functions are for reading and writing scheduling data via the JBin format:

```
import_schedule(<filename : String>)  
export_schedule(<filename : String>)
```

The google import/export functions deal with reading and writing Calendar data back and forth. The import function will display all Agile Planner tasks with a JSON format while the export function will schedule the tasks with timeslots according to the generated schedule (note: scheduled tasks are printed with links to their Calendar counterparts):

```
import_google()  
export_google()
```

These two functions are necessary when attempting to build a schedule with newly created tasks. The add_all_tasks() function deals with adding all task variables (whether past or current) to the schedule manager and the build() function creates and outputs a schedule:

```
add_all_tasks()  
build()
```

These functions allow you to visualize both the current Board setup, which comprises of all the Cards and their associated Tasks, as well as the generated schedule that was produced (either via 'build()' or from a prior session stored by JBin). The 'display_board' lets you see all the Cards and 'display_schedule' displays the created schedule:

```
display_board()  
display_schedule()
```

And finally, we arrive at one of the more interesting functions available with Simple. This operation allows the user to inject code while the script is being interpreted! Simply include whatever function calls, variable declarations, etc. as you'd like (making sure to close off with `__END__`). **Note: You cannot use inject_code() inside of custom functions or use it to create a function:**

```
inject_code()
```

Custom Functions:

Simple's custom functions are very similar to Python and allow for repeated efficiency and offer recursive capabilities. They follow this format here:

```
func <func_name>(<arg1>, <arg2>, . . .>)
```

A sample script is provided below as to what is possible outside of scheduling:

```
#####  
include: __CURR_CONFIG__, __LOG__  
# Outputs all binary codes of a specified length  
str: ""  
func binary(bin, x)  
    if(x.==(0))  
        str.concat(bin, "\n")  
        return  
    x.--()   
    binary(bin.add("0"), x)  
    binary(bin.add("1"), x)  
print("Enter number: ")  
x: input_int()  
binary("", x)  
write_file("data/bin.txt", str)  
#####
```

Control Structure:

If conditions work a bit differently compared to Python in that arguments are comma delimited (this is being changed with the next version of Simple script). Here is a typical example below:

```
if(x.==(0))  
    print("Play games with friends")  
elif(x.==(1))  
    print("Watch latest Marvel movie")  
else  
    print("Study for test")  
...
```

Object Methods:

Simple has an extensive list of methods for each type in order to leverage the Object-Oriented structure. A typical example would be as follows:

```
my_card: card("HW")
t1: task("Math", 4, 2)
# Adds the task to the card
my_card.add(t1)
```

While already sizable, the number of methods available continues to grow (you can see the current list via [Appendix B](#)).

System Logging:

System logging reports all actions that are performed with managing data and performing scheduling operations or routine IO. It is meant to be thorough and complete while avoiding unnecessary reporting. A sample log is shown below:

[06-05-2024] Log of all activities from current session:

```
[21:05:49] [INFO] CURRENT SESSION HAS BEGUN...
[21:05:49] [INFO] Reading Config: FILE=profile.cfg
[21:05:49] [INFO] USERNAME=null, EMAIL=null, WEEK_HOURS=[8, 8, 8, 8, 8, 8, 8], MAX_DAYS=14,
ARCHIVE_DAYS=14, PRIORITY=false, OVERFLOW=true, FIT_SCHEDULE=false, SCHEDULE_ALGO=1, MIN_HOURS=1
[21:05:50] [INFO] GOOGLE CALENDAR AUTHORIZATION PROCESSED...
[21:05:53] [INFO] SCRIPT_NAME=C:\Users\Student\Desktop\agile-planner\data\scripts\automate.smp1,
SCRIPT INSTANCE HAS BEGUN...
[21:05:56] [INFO] READ(JBIN): FILE=data/jbin/week.jbin
[21:05:56] [INFO] JBIN FILE PROCESSED
[21:06:14] [INFO] ADD(TASK): ID=0, NAME=CODMW2 w/ Sam, HOURS=4, DUE_DATE=05-06-2024
[21:06:14] [INFO] SCHEDULING HAS BEGUN...
[21:06:14] [INFO] DAY_ID=0, CAPACITY=8, HOURS_REMAINING=4, HOURS_FILLED=4, TASK ADDED=0,
OVERFLOW=false
[21:06:14] [INFO] DAY_ID=0, CAPACITY=8, HOURS_REMAINING=0, HOURS_FILLED=8, TASK ADDED=5,
OVERFLOW=false
[21:06:14] [INFO] DAY_ID=1, CAPACITY=8, HOURS_REMAINING=0, HOURS_FILLED=8, TASK ADDED=4,
OVERFLOW=false
[21:06:14] [INFO] DAY_ID=2, CAPACITY=8, HOURS_REMAINING=4, HOURS_FILLED=4, TASK ADDED=5,
OVERFLOW=false
[21:06:14] [INFO] DAY_ID=2, CAPACITY=8, HOURS_REMAINING=2, HOURS_FILLED=6, TASK ADDED=1,
OVERFLOW=false
[21:06:14] [INFO] DAY_ID=2, CAPACITY=8, HOURS_REMAINING=0, HOURS_FILLED=8, TASK ADDED=2,
OVERFLOW=false
[21:06:14] [INFO] DAY_ID=3, CAPACITY=8, HOURS_REMAINING=0, HOURS_FILLED=8, TASK ADDED=6,
OVERFLOW=false
```


[21:06:14] [INFO] DAY_ID=4, CAPACITY=8, HOURS_REMAINING=0, HOURS_FILLED=10, TASK_ADDED=2, OVERFLOW=true

[21:06:14] [INFO] DAY_ID=4, CAPACITY=8, HOURS_REMAINING=0, HOURS_FILLED=12, TASK_ADDED=6, OVERFLOW=true

[21:06:14] [INFO] SCHEDULING HAS FINISHED...

[21:06:18] [INFO] 7 TASKS REMOVED FROM GOOGLE CALENDAR...

[21:06:21] [INFO] SCHEDULE EXPORTED TO GOOGLE CALENDAR...

[21:06:35] [INFO] JBIN FILE CREATED

[21:06:35] [INFO] WRITE(JBIN): FILE=data/jbin/upcoming.jbin

[21:06:35] [INFO] SCRIPT_NAME=C:\Users\Student\Desktop\agile-planner\data\scripts\automate.smp1, SCRIPT_INSTANCE HAS ENDED...

Scripter Logging:

Scripter logging essentially serves as a stack trace and reports all operations that occur as the Simple script is parsed and interpreted. Changes are being planned with logging the following: variable creation, local stack, and global stack:

[06-05-2024] Log of all activities from current session:

```
[21:05:53] PREPROC_ATTR: DEF_CONFIG=false, LOG=true, STATS=false, HTML=true
[21:05:56] FUNC_CALLS: NAME=input_word, ARGS=["Import JBin -> "]
[21:05:56] VAR_SETUP: NAME=jbin_file, VALUE=week.jbin
[21:05:56] FUNC_CALLS: NAME=import_schedule, ARGS=[jbin_file]
[21:05:56] FUNC_SETUP: NAME= create_card, PARAM=[]
[21:05:59] FUNC_CALLS: NAME=input_bool, ARGS=["Create Card(T/F) -> "]
[21:05:59] VAR_SETUP: NAME=flag, VALUE=false
[21:05:59] IF_CONDITION: ARGS=[flag], RESULT=false
[21:05:59] IF_CONDITION: ARGS=[], RESULT=true
[21:05:59] FUNC_CALLS: NAME=println, ARGS=[]
[21:05:59] FUNC_CALLS: NAME=create_card, ARGS=[]
[21:05:59] FUNC_CALLS: NAME=add_all_cards, ARGS=[]
[21:05:59] FUNC_CALLS: NAME=display_board, ARGS=[]
[21:05:59] FUNC_SETUP: NAME= process_card, PARAM=[]
[21:05:59] CONST TYPE CREATED...
[21:05:59] VAR_SETUP: NAME=task_added, VALUE=false
[21:05:59] FUNC_SETUP: NAME= create_task, PARAM=[]
[21:06:01] FUNC_CALLS: NAME=input_bool, ARGS=["Create Task(T/F) -> "]
[21:06:01] VAR_SETUP: NAME=flag, VALUE=true
[21:06:01] IF_CONDITION: ARGS=[flag], RESULT=true
[21:06:01] VAR_SETUP: NAME=task_added, VALUE=true
[21:06:06] FUNC_CALLS: NAME=input_line, ARGS=["Name -> "]
[21:06:06] VAR_SETUP: NAME=_name, VALUE=CODMW2 w/ Sam
[21:06:07] FUNC_CALLS: NAME=input_int, ARGS=["Hours -> "]
[21:06:07] VAR_SETUP: NAME=_hours, VALUE=4
[21:06:09] FUNC_CALLS: NAME=input_int, ARGS=["Days -> "]
[21:06:09] VAR_SETUP: NAME=_days, VALUE=0
[21:06:09] VAR_SETUP: NAME=_t1, VALUE=Task [name=CODMW2 w/ Sam, total=4]
[21:06:12] FUNC_CALLS: NAME=input_int, ARGS=["Card Index -> "]
```

[21:06:12] VAR_SETUP: NAME=idx, VALUE=4
[21:06:12] ATTR_CALL: VAR_NAME=idx, NAME=<, ARGS[0]
[21:06:12] IF_CONDITION: ARGS=[idx.<(0)], RESULT=false
[21:06:12] IF_CONDITION: ARGS=[], RESULT=true
[21:06:12] FUNC_CALLS: NAME=get_card, ARGS=[idx]
[21:06:12] VAR_SETUP: NAME=c1, VALUE=Hobbies
[21:06:12] ATTR_CALL: VAR_NAME=c1, NAME=add, ARGS[_t1]
[21:06:12] FUNC_CALLS: NAME=process_card, ARGS=[_t1]
[21:06:14] FUNC_CALLS: NAME=input_bool, ARGS=["Create Task(T/F) -> "]
[21:06:14] VAR_SETUP: NAME=flag, VALUE=false
[21:06:14] IF_CONDITION: ARGS=[flag], RESULT=false
[21:06:14] IF_CONDITION: ARGS=[], RESULT=true
[21:06:14] FUNC_CALLS: NAME=println, ARGS=[]
[21:06:14] FUNC_CALLS: NAME=create_task, ARGS=[]
[21:06:14] FUNC_CALLS: NAME=create_task, ARGS=[]
[21:06:14] IF_CONDITION: ARGS=[task_added], RESULT=true
[21:06:14] FUNC_CALLS: NAME=display_board, ARGS=[]
[21:06:14] FUNC_CALLS: NAME=add_all_tasks, ARGS=[]
[21:06:14] FUNC_CALLS: NAME=build, ARGS=[]
[21:06:21] FUNC_CALLS: NAME=export_google, ARGS=[]
[21:06:21] FUNC_SETUP: NAME=if, PARAM=[]
[21:06:21] FUNC_CALLS: NAME=println, ARGS=["\n"]
[21:06:21] FUNC_CALLS: NAME=display_schedule, ARGS=[]
[21:06:35] FUNC_CALLS: NAME=input_word, ARGS=["Export JBin -> "]
[21:06:35] VAR_SETUP: NAME=jbin_file, VALUE=upcoming.jbin
[21:06:35] FUNC_CALLS: NAME=export_schedule, ARGS=[jbin_file]

Java Binary Serialization (JBin):

JBin for Agile Planner allows for efficient data persistence by working from the top down with data storage and thereby data retrieval. Its original inspiration was JSON and has thus proven capable so far with managing scheduling data of a wide variety. One stark difference compared to JSON is that it utilizes a basic pointer system for referencing any “owned” data types that reappear.

#####

06-05-2024

TASK {

Finish JBin, 4, -3

Finish Docs, 10, 0

Testing, 6, 2

Marketing, 4, -4

Study DP, 8, 0

Study Graphs, 8, 0

Apply for Jobs, 10, 2

}

CARD {

Default

Project, T0, T1, T2, T3

DS&A, T4, T5

Job Hunting, T6

Hobbies

}

DAY {

T6 8

T2 6, T6 2

}

#####

HTML Web Page Generation:

One core aspect that was greatly needed was a combined visualization of the script, system log, and script stack trace. With Agile Planner v0.4.0 [pre-release], we now have a new and improved option just for that:



The screenshot displays the 'Simple Script v0.4.0' web interface, which is divided into three main sections:

- automate.smpl:** Contains the source code for the script, including functions like `create_card()`, `process_card()`, `create_task()`, and `export_schedule()`.
- Script Log:** Shows a detailed log of script execution activities, including function calls, variable setups, and conditional checks with timestamps.
- System Log:** Displays system-level information such as session start, configuration reading, and file operations.

This allows you to store past sessions with ease via the 'html' folder in the agile-planner-0.4.0 app bundle.

Conclusive Summary v0.4.0:

As I was working on this version, I started to focus more on how we can make Agile Planner a more user-friendly experience. Before, my primary focus was on adding “cool” features that were useful but unfortunately not well integrated with the system. v0.4.0 really drove the idea of making this an actual application that was useful for both myself and others. I’m still continuing development and am planning on expanding the core system capabilities even further.

Conclusive Summary v0.3.0:

A lot of work has gone into making this all happen. The current plan is to continue developing the terminal version of Agile Planner until all core features hit the necessary mark. The current plan is to offer more 3rd party integrations, redesign the language, and implement optimization algorithms for scheduling.

If you have any questions or would like to report a bug, you can reach me via: aproe@ncsu.edu

Agile Planner – “Scheduling Made Simple”

Appendix A

```
# Adds all tasks to schedule manager (allows them to be scheduled)
add_all_tasks()

# Computes the average of provided values
avg(<arg1 : Integer | OPTIONAL>, <arg2 : Integer | OPTIONAL>, . . .) --> Integer

# Builds the schedule
build()

# Exports schedule to Google Calendar
export_google()

# Exports the schedule as a JBin file
export_schedule(<filename : String>)

# Gets a Card via a specified index (pairs well with the view_interface() function)
get_card(<index : Integer>) --> Card

# Imports schedule data from Google Calendar
import_google()

# Imports schedule from JBin file
import_schedule(<filename : String>)

# Allows user to inject additional code while script is being run
inject_code()

# Prompts user for Bool
input_bool(<prompt : String | OPTIONAL>) --> Bool

# Prompts user for Integer
input_int(<prompt: String | OPTIONAL>) --> Integer

# Prompts user for line of String
input_line(<prompt: String | OPTIONAL>) --> String

# Prompts user for number of Tasks
input_tasks(<index : Integer>)

# Prompts user for a String token or word
input_word(<prompt: String | OPTIONAL>) --> String

# Pauses script and waits for user to continue with prompt
pause()

# Prints out data without newline
print(<arg1 | OPTIONAL>, <arg2 | OPTIONAL>, . . .)
```

```
# Prints out data with newline
println(<arg1 | OPTIONAL>, <arg2 | OPTIONAL>, . . .)
# Sets the scheduling algorithm option
set_schedule(<index : Integer>)
# Views the Card interface from the scheduling manager
display_board()
# Views the global and local variable stack
display_stack()
# writes string contents to specified file
write_file(<filename : String>, <data : String>)
```


Appendix B

<In progress>