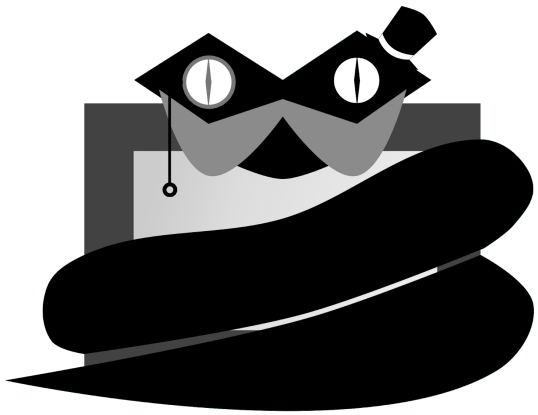
CHIP-GR8

REFERENCE MANUAL

2019

The CHIP-GR8 Team



DOCUMENT PURPOSE

This reference manual provides an introduction to the CHIP-GR8 API and the basics of CHIP-8 programming. The CHIP-GR8 API and associated CHIP-8 emulator are available through PIP using the command:

```
pip install chipgr8
```

Additional information on CHIP-GR8 can be found on the project's homepage at https://awiggs.github.io/chip-gr8/.

NOTE: this manual contains several examples of code which you can write in the interactive Chip-Gr8 display or in your own editor. You are highly encouraged to try these examples out.

CONTENTS

OVERVIEW

CHIP-GR8 is a library that aims to help developers create simple AI agents for CHIP-8 video games. The library is broken into two components, the CHIP-8 emulator, written in C, and the chipgr8 python module. The python module provides three interfaces for interacting with the CHIP-8. A command line interface, a python API, and a pygame display. These components are broken down in **FIGURE 1**.

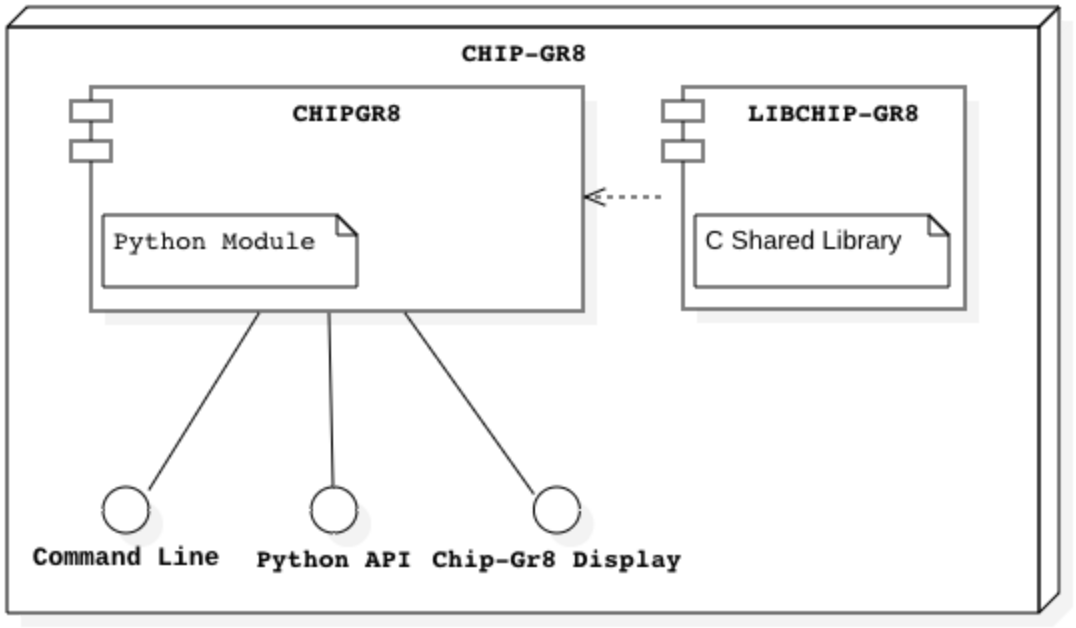


FIGURE 1: CHIP-GR8 Component Diagram

When first using CHIP-GR8 you will want to use the command line interface to get a sense of how games are played. Once you understand the visual interface learning how to perform the same interactions from code will be easier.

CHIP-8 Emulator

The CHIP-8 Emulator provided with CHIP-GR8 is written in C to increase performance and guarantee correct behaviour across platforms. You will never need to write or read C code to use

CHIP-GR8, however it is useful to understand the data structure and processing model that underlies the CHIP-8.

A CHIP-8 virtual machine instance is represented a single continuous strip of 0x1000 (4096) bytes. All CHIP-8 programs are stored as ROMs (binary files) that are loaded into memory starting at address 0x200. Execution begins at the same address. The first 0x200 (512) bytes are used to store all of the registers, the stack, and the video RAM of the CHIP-8. In the python module all of these fields are directly accessible from a VM instance. For example:

```
vm = chipgr8.init(ROM='pong')
vm.VM.RAM[0x200] # Retrieve the first byte of Pong
vm.VM.V[0xF]           # Read the value of register VF
vm.VM.I = 0x202   # Set address register I to 0x202
```

For information on the architecture and registers of the CHIP-8 see the **CHIP-8 REFERENCE** section.

> **NOTE:** While the assembler and disassembler of CHIP-GR8 support SUPER CHIP-8 instructions, the emulator currently does NOT.

Command Line Interface

CHIP-GR8 makes provides a straightforward command line interface to make jumping into a CHIP-8 game easy. If no arguments are provided to CHIP-GR8 when it is run as a module it will boot up a paused CHIP-8 virtual machine and CHIP-GR8 display. This can be done with the following command:

```
python -m chipgr8
```

Several additional options are provided and detailed below.

-h --help
This option will print usage information as well as short descriptions of each command line option. After printing this information CHIP-GR8 will immediately exit.

-V --version
This option will print the CHIP-GR8 version number and then immediately exit.

-v --verbose
This option will enable developer logging. The more v's provided the more logging. For example **-vvv** will provide the most possible logging.

-r <ROM> --rom <ROM>
This option allows you to provide the name or path to a ROM. by default CHIP-GR8 comes packaged with the following ROMS:

ROM	**AUTHOR**
15 Puzzle	Roger Ivie
Animal Race	Brian Astle
Astro Dodge	Revival Studios
Blinky	Hans Christian Egeberg
Breakout	David Winter
Brix	Andreas Gustafsson
Hidden	David Winter
Lunar Lander	Udo Pernisz
Pong (1 player)	Unknown
Pong (2 player)	Paul Vervalin
Space Invaders	David Winter
Squash	David Winter
Tetris	Fran Dachille
Worm	Revival Studios

-a <SOURCE> --assemble <SOURCE>
Assembles the provided source file. By default the assembled source will be printed to stdout, but if the **--out** option is provided the output will instead be written to that file.

-d <BINARY> --disassemble <BINARY>
Disassembles the provided binary file. By default the disassembled source will be printed to stdout, but if the **--out** option is provided the output will instead be written to that file

-o <OUT> --out <OUT>
Provide an output file for the assembler/disassembler.

-L --no-labels
If provided when disassembling the disassembler will not try to intelligently place labels, instead all **JP** and **CALL** instructions will explicitly provide the address being moved to.

-s --smooth
Enables the experimental smooth rendering mode. This is slow on most machines, but can decrease flashing.

-f <FOREGROUND> --foreground <FOREGROUND>
Specify a foreground color for the CHIP-GR8 display. This value should be provided as either a 3 or 6 digit hexadecimal code, for example FAFE00.

-b <BACKGROUND> --background <BACKGROUND>
Specify a background color for the CHIP-GR8 display. This value should be provided as either a 3 or 6 digit hexadecimal code, for example FAFE00

-t <THEME> --theme <THEME>
Specify one of the pre-existing themes for the CHIP-GR8 display. This option is overwritten by the **--foreground** and **--background** options. The following themes are currently included: light, dark, sunrise, hacker, redalert, and snake.

-S --no-scroll
If provided the CHIP-GR8 will not auto-scroll and highlight when games are running. This can be helpful for users who want less information flashing on their screen.

Python API

CHIP-GR8 provides a python module and API under the name **chipgr8.** This module provides several utilities for working with ROMs, feature

detection in games, and creating CHIP-8 virtual machines. The bread and butter command off the API is **init**, which is how you instantiate VM instances. This command takes many options to help you precisely configure your vm.

For an example of using the API see **WRITE AN AI AGENT** section of this manual. For a detailed outline of the API see the **API REFERENCE**.

CHIP-GR8 Display

The CHIP-GR8 display is the easiest way to interact with a CHIP-8 vm. It allows you to play games, watch your AI agent perform, and query VM state to discover new features. The CHIP-GR8 display is described in detail in the **DISPLAY** section of this manual.

DISPLAY

CHIP-GR8 offers an interactive display for playing and testing games as well as watching AI agents perform. When you start CHIP-GR8 from the command line the display is automatically opened. If you want to open the display from the API pass **display=True** to the **init** function.

All of the keybindings described in this section are the default keybindings for CHIP-GR8. To change your keybindings you can create a **chipgr8.keys.json** in your local directory and provide your own key bindings. Default key bindings are available programmatically through:

```
chipgr8.defaultBindings
```

When you open the display it should look something line in **FIGURE 2.**

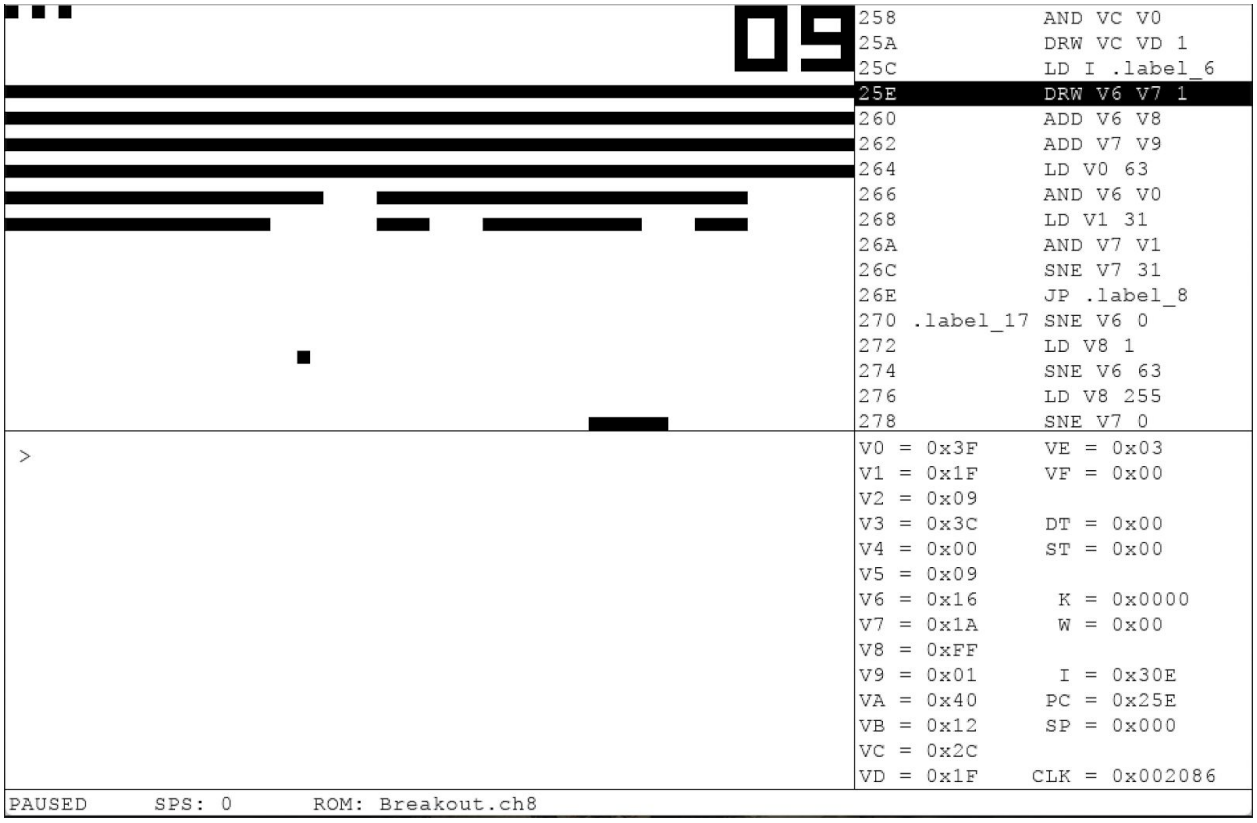


FIGURE 2: CHIP-GR8 Display

To exit the window hit the close button on your OS window or hit the **ESC** key on your keyboard. Several themes are available for the display, these can either be set using the appropriate command line argument or programmatically by passing **theme=<theme>** or **foreground=<color>** or **background=<color>** to the **init** function.

The display is broken into 5 main panels:

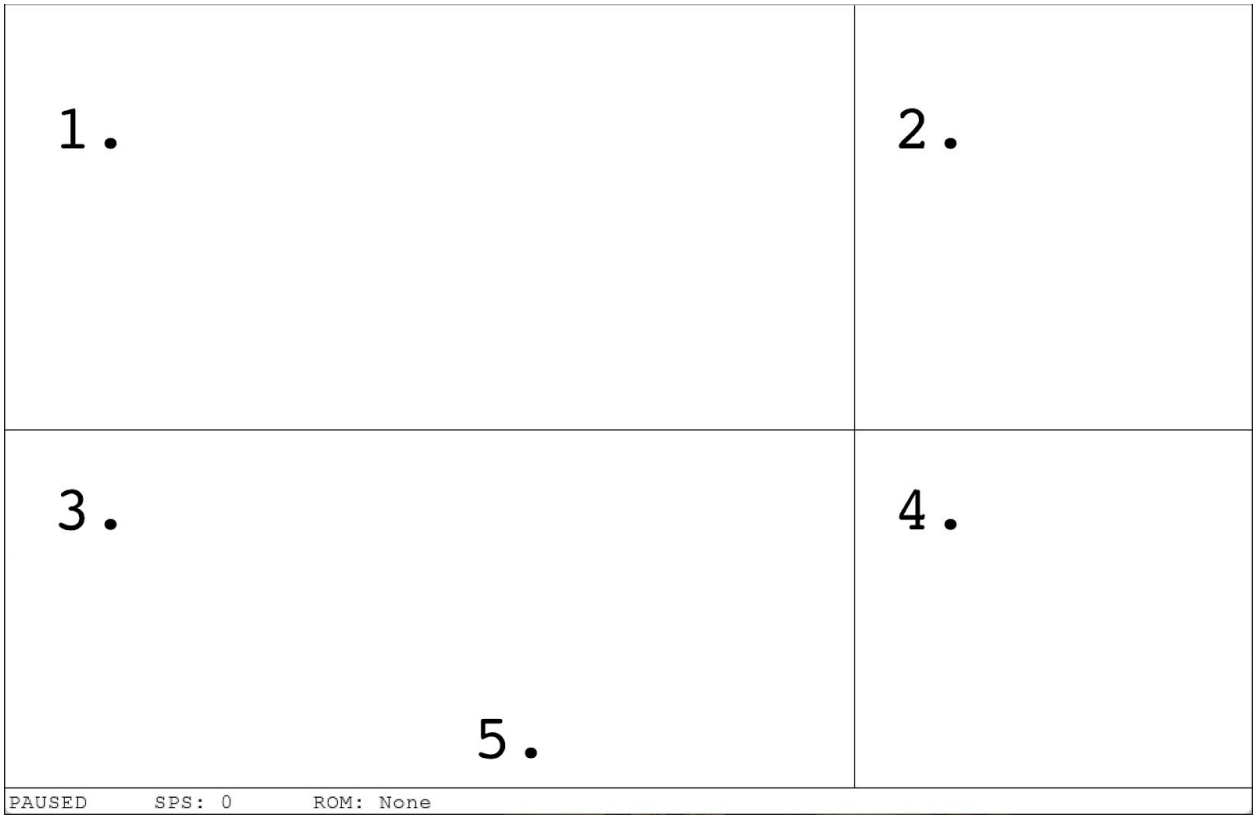


FIGURE 3: Labelled CHIP-GR8 Display

1. Game Display

The first panel of the display shows the CHIP-GR8 game as a 64x32 grid of pixels. Due to the way CHIP-GR8 games draw to the screen most sprites flash on and off. This effect is normal. To control the game you can use the following keys to press the corresponding CHIP-8 keys:

```
1 2 3 4      1 2 3 C
q w e r  ->  4 5 6 D
```

```
a s d f      7 8 9 E
z x c v      A 0 B F
```

Every game uses slightly different control schemes, but often **q**, **2**, **e**, and **s** are used for the four cardinal directions left, up, right, and down. The **v** key is often the start button for games.

2. Disassembly Source

This panel shows the disassembly source of the ROM. By default the disassembly will scroll with each step of the CHIP-8. The currently executing instruction will be highlighted. When you pause the CHIP-8 by hitting **F5** the disassembly will also pause. When paused you can scroll the disassembly with your mouse, **PageUp**, **PageDown**, **Home**, and **End**. To step through the instructions you can press or hold **F6**.

NOTE: There are several limitations to the CHIP-GR8 live disassembler.

The CHIP-8 is capable of writing code on the fly by storing data to program memory, but for performance reasons the live disassembly is only generated once. This means ROMs that use this technique will appear incorrect in the live disassembly.

Some ROMs make use of instructions at odd and even addresses. The CHIP-GR8 disassembler uses several heuristics to guess how instructions should be decoded, but sometimes will guess wrong or fail and resort to simply producing 'BYTE 0xXX'. This is especially common if sprite data is intermingled with program data. Be aware of these limitations if the live disassembly appears off.

3. Interactive REPL

The interactive REPL evaluates and executes python expressions and statements. The CHIP-GR8 python module is made available through the module variable **chipgr8** as well as the python **math** module. The current CHIP-9 VM can be accessed through the **vm** variable. Several common commands are also aliased for convenience.

COMMAND	**DESCRIPTION**
help	Display short help message listing these commands.

play()	Resume the CHIP-8 vm.
reset()	Reset the CHIP-8 vm, resetting all of RAM and reloading the current ROM.
step()	Step the CHIP-8 VM forward one step.
Query()	Create a new memory Query object that queries the current CHIP-8 VM instance.
loadROM(rom)	Loads **rom** into the current CHIP-8 VM instance.
addBreakpoint(addr)	Adds a breakpoint for when the program counter hits **addr**. This will automatically pause the CHIP-8 VM.
removeBreakpoint(addr)	Removes a breakpoint at **addr**.
toggleBreakpoint(addr)	Toggles a breakpoint at **addr**.
clearBreakpoints()	Removes all current breakpoints.
read(file)	Returns the content of **file** as a string.
write(file, content)	Write **content** to **file**.

The REPL has several limitations. Input must be a single line less than or equal to 60 characters in length. Additionally, long output from functions will be wrapped if they exceed the terminals line length.

4. Live Registers

This view provides the live values of the CHIP-8 registers. Many of these registers have short cryptic names. Full descriptions of each can be found in the **CHIP-8 REFERENCE** section of this manual.

These registers are extremely useful when debugging ROMs and discovering features. In particular, the **K** register represents the current input to the VM instance. This can be useful if your AI appears to not be doing anything or you want to encode the input for a new ROM.

NOTE: If you find that you are unable to interact with games ensure

that the **K** register is changing when you press keys. If it is not you probably have not set **aiInputMask** to an appropriate value. If this field is 0 you will be able to press all keys, whereas a value of 0xFFFF will not allow any of your input through. 0xFFFF is the default value when using the API. You can set **aiInputMask** from the REPL with:

```
    vm.aiInputMask = 0x0
```

Or from the API with:

```
    chipgr8.init(aiInputMask=0x0)
```

5. Status Bar

The status bar provides a quick look at basic information. This includes whether the CHIP-8 vm instance is PLAYING or PAUSED, the current Steps Per Second (SPS), and the currently loaded ROM. The SPS will try and hit the **frequency** you initialized the VM instance with, by default 600Hz, but on most machines this value will vary significantly.

WRITING AN AI AGENT

This example will help you write your first AI agent. An AI agent performs two key tasks, observations and actions. To get started import the CHIP-GR8 API, and a game object. The game objects provide useful defaults for observations, actions, as well as the correct ROM name. For this example we will use the game Squash. Squash is a single player Pong. We will also import random for later.

```
import random
import chipgr8
from chipgr8.games import Squash
```

AI agents are trained and run in loops. This is typically done with a while loop where you wait for a VM instance to be done. For our first agent lets just pick a random action. In order to run this agent we will need to create a VM instance to run it on and load the Squash ROM.

```
vm = chipgr8.init(ROM=Squash.ROM)
```

By default the API returns a vm appropriate for running a single AI. we will now create a loop where we repeatedly choose a random action. **Squash.actions** provides a list of all the valid Squash game actions. We also need to indicate when the VM instance should be considered done. The Squash object also provides this in its set of observations, so we will observe the VM and check to see if the VM is done.

```
while not vm.done():
    vm.act(random.choice(Squash.actions))
    observations = Squash.observe(vm)
    vm.doneIf(observations.done)
```

Our AI will now run, but we will not be able to see it perform any of its actions. We can watch a replay using the **inputHistory** of our vm. The **go** method will loop the vm instance for us.

```
chipgr8.init(
    ROM=Squash.ROM,
    inputHistory=vm.inputHistory,
    display=True
).go()
```

Our AI is not very good, but we can easily make it better just by running multiple random AI agents and picking the best one. Let's start by creating 100 vm instances.

```
vms = chipgr8.init(ROM=Squash.ROM, instances=100)
```

We can now iterate over the vms and run each one like we did before.

```
while not vms.done():
    for vm in vms:
        vm.act(random.choice(Squash.actions))
        observations = Squash.observe(vm)
        vm.doneIf(observations.done)
```

This approach is a little slow though since we have to run ever VM instance as part of the same process. We can take advantage of a machines multiple cores by using the vms **inParallel()** method. This method requires us to refactor our code a little bit. This method expects a function which will be called repeatedly until vm instance is done. We can do this by taking our inner section of the loop and turning it into a function.

```
def action(vm):
    vm.act(random.choice(Squash.actions))
    observations = Squash.observe(vm)
    vm.doneIf(observations.done)

vms.inParallel(action)
```

We can now just pick the best vm of the bunch. The squash Object thankfully has another observation that can help us, score. We can use the vms **maxBy** function to get the best VM.

```
best = vms.maxBy(lambda vm : Squash.observe(vm).score)
```

We can now watch this vm like we did before using its **inputHistory.** Congratulations on writing your first CHIP-GR8 AI agent!. You can find the final code altogether below.

```
import random
import chipgr8
from chipgr8.games import Squash

# This action is performed repeatedly until the VM is done
def action(vm):
    vm.act(random.choice(Squash.actions))
    observations = Squash.observe(vm)
    vm.doneIf(observations.done)

# Create 100 CHIP-8 VM instances
vms = chipgr8.init(ROM=Squash.ROM, instances=100)
# Run all our random agents
vms.inParallel(action)
# Pick the best one
best = vms.maxBy(lambda vm : Squash.observe(vm).score)

# Show a replay of the best
chipgr8.init(
    ROM=Squash.ROM,
    inputHistory=best.inputHistory,
    display=True
).go()
```

QUERYING MEMORY

In order to support more games, or find additional values from CHIP-8 RAM for games already included with CHIP-GR8, components are provided for querying memory. These components are meant to be used in a workflow like the following:

1. Start the CHIP-GR8 display with the ROM you want to query.
2. Put the VM into a state you understand.
3. Create a **Query** object and use a predicate to limit the number of matches memory addresses.
4. Change the VM to a new state and use a new predicate to further filter the results.
5. Repeat step 4 until there is only a single address that matches.
6. Copy the **Query** out to a file.

Several steps are made easier by the fact that **Query** and **Observer** objects will print their own source code in the REPL. You can easily write these to a file using the **write** function.

Queries

Queries provide several predicates to limit discover memory addresses, like **eq()**, **dec()**, **lte()**, etc. A full list and description can be found in the **API REFERENCE** section of this manual.

A list of all memory addresses, along with their previously queried values can be found using the **previous** field. For example:

```
q = Query(vm)
q.eq(0x04)
print(q.previous)
```

You can instantiate a finished **Query** by providing an address instead of a VM instance. For example, to create a query that looks at address 0x200:

```
q = Query(0x200)
```

This **Query** can now be used to retrieve the value in CHIP-8 RAM at 0x200 of any VM instance with:

```
q.observe(vm)
```

Observers

Queries can be combined using an **Observer.** An observer is just a collection of queries and functions that provides one method, **observe**, which applies all these queries and functions to a provided VM instance and returns the result as a **NamedList**, a data structure that behaves like a python list, but can be accessed by attributes and keys, for example to create a list of one element with a key 7:

```
myNamedList = NamedList(['key'], [7])
```

To access the element you can use the following ways:

```
myNameList[0]     # By index
myNameList.key    # By attribute
myNameList['key'] # By key
```

To add queries to an observer you can call **addQuery**. This method also accepts functions that take two arguments. The first argument is all non-function observations. The second is the VM instance. This allows you to create combinational queries. For example

```
o = Observer()
o.addQuery('lives', Query(0x115))
o.addQuery('done',  lambda o, vm : o.lives == 0)
```

Games

Games provide actions, observations, and a ROM all in one package. Several games are provided out of the box, but you can also create your own game objects for ROMs not included with CHIP-GR8.

ASSEMBLER & DISASSEMBLER

CHIP-GR8 comes equipped with a CHIP-8 assembler and disassembler. You can use it to assemble and disassemble CHIP-8 ROMSs from the command line and from your own python files. For example to assemble a short ROM:

```
chipgr8.assemble('''
    ; Displays 42 in the top right corner
        ld v0 4
        ld v1 2
        ld v2 1
        ld v3 1
        ld f v0
        drw v2 v3 5
        add v2 5
        ld f v1
        drw v2 v3 5
    .done
        jp .done
''', outPath='myROM.ch8')
```

Assembled ROMs can be disassembled. While most information is lost after assembly the disassembler does attempt to guess label positions for **JMP** and **CALL** instructions. You can disassemble this same ROM with:

```
print(chipgr8.disassemble(inPath='myROM.ch8'))
```

There are several options when disassembling. These are detailed in the **API REFERENCE** section of this manual.

The CHIP-8 assembler accepts all standard CHIP-8 instructions. All instructions are listed and detailed in the **CHIP-8 REFERENCE** section of this manual. Instructions are NOT case sensitive so the following are equivalent.

```
LD V0 4
ld v0 4
```

The CHIP-8 assembler syntax has several additional features to make it easier to write and read ROM sources.

Literals

You can provide numeric literals to the assembler in decimal (default) hexadecimal or binary formats. The latter require the prefixes 0x and 0b respectively. For example the following three lines are equivalent:

```
ld v0 15
ld v0 0xF
ld v0 0b1111
```

Comments

Comments are lines of code that will be ignored by the assembler, they can appear on the same line as valid code. Comments must begin with a semicolon. For example:

```
; This is a comment
ld v0 4 ; This is also a comment
```

Labels

Labels are sections of code that can be jumped to, called, or loaded from. Labels get automatically converted to a memory address when assembled. All labels must begin with a period and contain no whitespace. For example:

```
.start
    ld v0 4
    ld I .sprite
    jp .start
.sprite
    byte 0xFF
    byte 0xF
```

Constants

Constants provide a way of aliasing registers and declaring numeric values. Constants may also refer to other constants and labels. To declare a constant simply start a line with one followed by its

value. It can then be used in place of any instruction argument.Constants must begin with the dollar sign character and contain no whitespace. Constants are particularly useful when referring to general purpose registers by their desired purpose. For example:

```
$lives v0
$startingLives 5
$start .start
    call $start
.done
    jp .done
.start
    ld $lives $startingLives
```

Bytes

For sprites and data you may want to insert byte data directly into your ROM. This can be done with the **BYTE** pseudo-instruction. This pseudo-instruction takes a single argument, a one byte numeric value that will be inserted into the ROM at the current location. For example:

```
.spritePlayer
    byte 0b11000000
    byte 0b11000000
    byte 0b11000000
    byte 0b11000000
.spriteBall
    byte 0b11000000
    byte 0b11000000
```

ACKNOWLEDGEMENTS

CHIP-GR8 is only possible because of work by several people before us. We would like to thank the following people and efforts for inspiring and helping us create this library.

CHIP-8 design reference:
http://devernay.free.fr/hacks/chip8/C8TECH10.HTM#2.4

Display design reference:
https://github.com/massung/CHIP-8

AI API reference:
https://gym.openai.com/

Reference manual inspiration:
http://www.zachtronics.com/tis-100/

ROM authors:
Andreas Gustafsson
Brian Astle
David Winter
Fran Dachille
Hans Christian Egeberg
Paul Vervalin
Revival Studios
Roger Ivie
Udo Pernisz

CHIP-8 creator:
Joseph Weisbecker

API REFERENCE

---

assemble(source=None, inPath=None, outPath=None)

Converts assembly **source** code, or source code contained in **inPath** into binary data (a ROM). This ROM may optionally be written to file with the **outPath** argument.

Chip8VM (Class)

Represents a CHIP-8 virtual machine. Provides interface and controls for display and input. Rather than initializing directly, an instance of this class or its sister class **Chip8VMs** should always be instantiated using **init**.

(Fields)

aiInputMask

A number that controls what keys are usable by AI agents calling **act** and what keys are usable by a user on their keyboard. For example, An **aiInputMask** of 0x0000 will prevent an AI agent from using any keys, but a user will be able to use all keys.

inputHistory

A list of number pairs that represent changes in key presses. The first value in the pair is the key value, the second is the clock value when input changed to that value.

paused

A control flag set to True if the display is paused.

pyclock

The pygame clock used to keep track of time between steps when using the CHIP-GR8 display.

record

A control flag set to True if **inputHistory** is being recorded.

ROM

The path to the currently loaded game ROM.

sampleRate

The number of steps that are performed when an AI calls **act**.

smooth

A control flag for the experimental smooth rendering mode. This mode is slow on most machines.

speed

The speed at which the UI is tied to the CHIP-8 frequency. So when speed is 1, games will appear to run at the provided freq, but when speed is 2, games will appear to run twice as fast. Must be provided as an integer

> **NOTE:** Speed is heavily limited by your machine's performance, so speeds above 1 will likely be limited by your maximum framerate.

VM

A direct reference to the CHIP-8 c-struct. This provides direct memory access (eg. **VM.RAM[0x200]**) as well as register reference (eg. **VM.PC**). Use these fields with caution as inappropriate usage can result in a segmentation fault. Direct references to **VM** should not be maintained (no aliasing).

(Methods)

addBreakpoint(addr)

Add a breakpoint at **addr**. When the VM steps to this address (when PC is equal to **addr**) the CHIP-GR8 display will automatically pause.

removeBreakpoint(addr)

Remove a breakpoint at **addr**.

toggleBreakpoint(addr)

Toggles a breakpoint at **addr**.

clearBreakpoints()

Clear all current breakpoints.

act(action, repeat=1)

Allows an AI agent to perform **action** (action is an input key value) and steps the CHIP-8 emulator forward **sampleRate** clock cycles. This action will be repeated **repeat** times.

actUntil(action, predicate)

Performs **act(action)** in a loop until the provided predicate returns true. The predicate is called with the VM instance.

ctx()

Returns an instance of the CHIP-8's VRAM in a numpy compliant format (lazyarray). Pixel values can be addressed directly. (eg. a pixel at position (16, 8) can be retrieved with **ctx()[16, 8]**). This method is safe to call repeatedly.

done()

Returns True if the VM is done and has NOT been reset.

doneIf(done)

Signals to the VM that it is done.

go()

Starts the VM in an until **done()** loop, calling **act(0)** repeatedly. This is ideal for user interaction without an AI agent.

input(keys)

Send an input key value to the CHIP-8 emulator. Input keys are masked by **aiInputMask**.

loadROM(nameOrPath, reset=True)

Loads a ROM from the provided path or searches for the name in the set of provided ROM files. If **reset** is True the VM will be reset prior to loading the ROM.

loadState(path=None, tag=None)

Load a CHIP-8 emulator state from a **path** or by associated **tag**, restoring a previous state of **VM**.

saveState(path=None, tag=None, force=False)

Save the current CHIP-8 emulator state to a **path** or **tag**. If **force** is True files will be overwritten.

reset()

Reset the VM with the current ROM still loaded.

step()

Step the VM forward 1 clock cycle.

Chip8VMs (Class)

Represents a collection of CHIP-8 virtual machines. Provides an interface for interacting with and filtering several virtual machines at the same time. This class is iterable, and will iterate over all vms that are NOT **done()**.

(Methods)

done()

Returns True if all VM instances are done.

find(predicate)

Find a specific VM using a function **predicate** that takes a VM as an argument and returns True or False. Returns the first VM for which the **predicate** was True. Searches done and not done vms.

inParallel(do)

Performs a function **do** on all not done vms in parallel. The function is expected to take the VM as an argument. When using this method external VM references can become out of date due to pickling across processes.

maxBy(projection)

Returns the VM with the maximum value by the given **projection**, a function that takes a VM as its argument and returns a comparable value.

minBy(projection)

Returns the VM with the minimum value by the given **projection**, a function that takes a VM as its argument and returns a comparable value.

reset()
Resets all the vms.

defaultBindings
Default key bindings for the CHIP-GR8 display as a python dictionary.

disassemble((Parameters))
Converts a binary ROM into an assembly source file. Returns the source. Provides option for disassembling with labels and special formats.

(Parameters)

buffer=None
The binary ROM to disassemble as a set of bytes. Optional if **inPath** is provided.

inPath=None
The path to a binary ROM to disassemble. Optional if **buffer** is provided.

outPath=None
If the path is provided, the source code is written to that file.

labels={}
A dictionary used to generate labels. If None is passed, labels will not be generated in the source.

decargs=True
If True, instruction numerical operands will be output in decimal rather than hexadecimal.

srcFormat='{label}{labelSep}{prefix}{instruction}\n'
A format string for lines of source code. Can contain the following variables **label**, **labelSep**, **prefix**, **instruction**, **addr**, and **dump**. For example for hexdump with address use:

```
srcFormat='{addr} {dump}'
```

labelSep = '\n '

The string used to separate labels from instructions.

prefix=' '
The string used to prefix all instructions.

addrTable={}
A table that will have addresses as keys and instructions as values.

findROM(rom)
Returns the path to **rom** if it is one of the included ROMs.

Game (Class)
A generic class for game specific data. Game specific instances of this class exist for each included ROM.

(Fields)

actions
A list of valid actions (key values) for the given game.

ROM
The name of the ROM file for this game.

(Methods)

observe(vm)
Returns a set of game specific observations given a vm.

hexdump(buffer=None, inPath=None, outPath=None)
Dumps a **buffer** or file at **inPath** as a set of 16bit hexadecimal values on each line (the number of bits that correspond to a CHIP-8 instruction). Writes the data to **outPath** if provided.

init((Parameters))
Returns an instance of **Chip8VM** or **Chip8VMs**. Used to configure the virtual machines for a user or a given AI agent.

(Parameters)

ROM=None
If provided will load a ROM into the VM instance or instances.

frequency=600
The starting **frequency** of the VM instance or instances. Will automatically be set to the closest multiple of 60 less than or equal to the provided **frequency**.

loadState=None
A path or tag to a VM save state that will be loaded into each VM instance or instances.

inputHistory=None
If provided user and AI input will be ignored and the history will be used to reproduce the same events.

sampleRate=1
The number of steps that are performed when an AI calls **act**.

instances=1
The number of VM instances to create.

display=False
If True, the VM will create a CHIP-GR8 display. Cannot be True if instances does not equal 1.

smooth=False
If True, enables the experimental smooth rendering mode. This mode is slow on most machines.

startPaused=False
If True, the VM instance will start paused.

aiInputMask=0xFFFF
The keys usable to the AI agent as a bitmask. The keys available to the user are the bitwise inverse of this mask.

foreground=(255, 255, 255)
The foreground color of the CHIP-GR8 display as an RGB tuple or hex code.

background=(0, 0, 0)

The background color of the CHIP-GR8 display as an RGB tuple or hex code.

theme=None,
The foreground/background color provided as a tuple.

autoScroll=True
If True, this disassembly source will automatically scroll when the CHIP-GR8 display is open and a ROM is running.

speed
The speed at which the UI is tied to the CHIP-8 frequency. So when speed is 1, games will appear to run at the provided freq, but when speed is 2, games will appear to run twice as fast. Must be provided as an integer.

NamedList (Class)
A list-like structure that allows elements to be accessed by named properties. Behaves like a python list, can be iterated, indexed, spliced, and measured with **len()**.

(Fields)

names
A list of keys for the list in order.

values
A list of values for the list in order.

(Methods)

append(name, value)
Append a **name** and **value** to the list.

nparray()
Retrieve the values of the list as an numpy ndarray.

tensor()
Retrieve the values of the list as a tensorflow tensor.

Observer (Class)
Represents a collection of queries that can be applied to a VM acquiring a set of observations.

(Methods)

addQuery(name, query)

Add a query with an associated name to an observer. Accepts either a finalized query or a function that accepts a set of observations (**NamedList**) as the first argument and a VM instance as its second argument. This function argument can be used to create compound queries.

observe(vm)

Retrieve a set of observations as a **NamedList** given a **vm** instance.

Query (Class)

Used to find a specific memory address. When using a query to search for a memory address, several predicates can be used to filter the query.

(Fields)

done

True if the query has found 0 or 1 addresses.

Success

True if the query has found 1 address.

(Methods)

dec()

Filter queiried memory addresses by values that have decreased since the last query.

eq(value)

Filter queiried memory addresses by values that equal **value.**

gt(value)

Filter queiried memory addresses by values that are greater than **value.**

gte(value)

Filter queiried memory addresses by values that are greater than or equal to **value**.

inc()

Filter queiried memory addresses by values that have increased since the last query.

lt(value)

Filter queiried memory addresses by values that are less than **value**.

lte(value)

Filter queiried memory addresses by values that are less than or equal to **value**.

observe(vm)

If a query is successful this method returns the value at the VM instance's RAM corresponding to this query.

unknown()

Refresh the previous values of all currently queried memory addresses.

readableInputHistory(inputHistory, names)

Given an **inputHistory** and a set of actions, **names**, as a **NamedList**, produces a human readable version of the **inputHistory**.

Themes

A python dictionary of the builtin CHIP-GR8 themes.

CHIP-8 REFERENCE

(**Registers)**

V0, V1, ..., VF
16 general purpose registers. Each can store an 8-bit unsigned integer. **VF** is used as a general purpose flag register for arithmetic and other instructions.

DT
The delay timer register is an 8-bit unsigned integer that is automatically decremented at 60Hz until its value is 0.

ST
The sound timer register is an 8-bit unsigned integer that is automatically decremented at 60Hz until its value is 0. Whenever **ST** is greater than 0 the CHIP-GR8 display will produce a pure tone.

K
The keys register is a 16-bit unsigned integer where each bit of the field represents a key on the CHIP-8 keyboard.

W
The wait register is an 8-bit unsigned integer that is used to indicate which general purpose register to store the next key press into.

I
The address register is a 16-bit unsigned integer used to obtain values from CHIP-8’s RAM.

PC
The program counter is a 16-bit unsigned integer used to indicate which instruction the CHIP-8 is currently executing.

SP
The stack pointer is an 8-bit unsigned integer used to indicate the head of the address stack.

CLK
The clock register is a 64-bit unsigned integer used to indicate the number of cycles that have occurred since thee CHIP-8 interpreter began running.

(Instructions)

> **NOTE:** Some instructions are expected to behave differently for different ROMs. This behaviour is captured in a **quirks** field of the VM struct. This field can be set with appropriate flags to enable different instruction behaviour. For example:
>
> ```
> vm = chipgr8.init(ROM='myROM')
> vm.VM.quirks = 0x01
> ```

CLS
Clears the display (sets all of VRAM to 0).

RET
Returns from a subroutine. Sets **PC** to the address on top of the stack and then decrememements **SP** (pops the stack).

JP <ADDR>
Sets **PC** to <ADDR>.

CALL <ADDR>
Increments **SP**, puts the current **PC** on top of the stack and sets **PC** to <ADDR>.

SE <Vx> <BYTE>
Skips the next instruction if the value stored in general purpose register <Vx> is equal to <BYTE>.

SNE <Vx> <BYTE>
Skips the next instruction if the value stored in general purpose register <Vx> is NOT equal to <BYTE>.

SE <Vx> <Vy>
Skip the next instruction if the value stored in general purpose register <Vx> is equal to the value stored in general purpose register <Vy>.

LD <Vx> <BYTE>
Immediately load the value <BYTE> into general purpose register <Vx>.

ADD <Vx> <BYTE>
Takes the unsigned sum of the values stored in general purpose registers <Vx> and <Vy>, then stores the result in <Vx>.

LD <Vx> <Vy>
Stores the value stored in general purpose register <Vy> in general purpose register <Vx>.

OR <Vx> <Vy>
Takes the bitwise OR of the values stored in general purpose registers <Vx> and <Vy>, then stores the result in <Vx>.

AND <Vx> <Vy>
Takes the bitwise AND of the values stored in general purpose registers <Vx> and <Vy>, then stores the result in <Vx>.

XOR <Vx> <Vy>
Takes the bitwise XOR of the values stored in general purpose registers <Vx> and <Vy>, then stores the result in <Vx>.

ADD <Vx> <Vy>
Takes the unsigned sum of the values stored in general purpose registers <Vx> and <Vy>, then stores the result in <Vx>. If the result overflows **VF** is set to 1, otherwise it is set to 0.

SUB <Vx> <Vy>
Takes the unsigned difference of the values stored in general purpose registers <Vx> and <Vy>, then stores the result in <Vx>. **VF** is set to 1 if <Vx> > <Vy>, otherwise it is set to 0.

SHR <Vx>
Right shift the value stored in general purpose register <Vx>, then stores the result in <Vx>. **VF** is set to 1 if the least significant bit of <Vx> is 1, otherwise it is set to 0.

> **NOTE:** Some ROMs expect this instruction to have the signature **SHR <Vx> <Vy>.** In this case the value of <Vy> will be shifted like <Vx> in the original instruction and stored in <Vx>. This functionality can be enabled by ORing the quirk field with 0x01.

SUBN <Vx> <Vy>
Takes the unsigned difference of the values stored in general purpose registers <Vy> and <Vx>, then stores the result in <Vx>. **VF** is set to 1 if <Vx> > <Vy>, otherwise it is set to 0.

SHL <Vx>
Left shift the value stored in general purpose register <Vx>, then stores the result in <Vx>. **VF** is set to 1 if the least significant bit of <Vx> is 1, otherwise it is set to 0.

> **NOTE:** Some ROMs expect this instruction to have the signature **SHL <Vx> <Vy>**. In this case the value of <Vy> will be shifted like <Vx> in the original instruction and stored in <Vx>. This functionality can be enabled by ORing the quirk field with 0x01.

SNE <Vx> <Vy>
Skip the next instruction if the value stored in general purpose register <Vx> is NOT equal to the value stored in general purpose register <Vy>.

LD I <ADDR>
The value of register **I** is set to <ADDR>.

JP V0 <ADDR>
Set **PC** to the sum of <ADDR> and the value stored in general purpose register **V0**.

RND <Vx> <BYTE>
Takes the bitwise AND of a random value and <BYE>, then stores the result in <Vx>.

DRW <Vx> <Vy> <NIBBLE>
Draw an 8 by <NIBBLE> sprite at x position stored in general purpose <Vx> and y position <Vy>. <NIBBLE> bytes sprite data is read from the memory address stored in register **I**. Data is XORed onto the display. If any bit flips from 1 to 0 **VF** is set to 1, otherwise it is set to 0. This indicates a collision.

SKP <Vx>
Skip the next instruction if the key with the value stored in general purpose register <Vx> is currently down.

SKNP <Vx>
Skip the next instruction if the key with the value stored in general purpose register <Vx> is not currently down.

LD <Vx> DT
The value of register **DT** is stored in general purpose register <Vx>.

LD <Vx> K
The value of register **K** is stored in general purpose register <Vx> as soon as the next key is pressed.

LD DT <Vx>
The value of register **DT** is set to the value stored in general purpose register <Vx>.

LD ST <Vx>
The value of register **ST** is set to the value stored in general purpose register <Vx>

ADD I <Vx>
Takes the unsigned sum of the values stored in general purpose register <Vx> and register **I**, then stores the result in **I**.

LD F <Vx>
The value of register **I** is set to the address of the numerical sprite that represents the value stored in general purpose register <Vx>.

LD B <Vx>
Stores the hundreds digit of the value stored at general purpose register <Vx> at the address stored in register **I**, the tens digit at the address **I** + 1, and the ones digit at the address **I** + 2.

LD [I] <Vx>
Store the values of general purpose registers **V0**, **V1**, ..., <Vx> in memory locations starting at the address stored in register **I**.

NOTE: Some ROMs expect this instruction to increment the value stored in register **I** as each general purpose register is stored. This functionality can be enabled by ORing the quirk field with 0x02.

LD <Vx> [I]

Reads the values starting at the address stored in register **I** into general purpose register **V0, V1, ...,** <Vx>.

> **NOTE:** Some ROMs expect this instruction to increment the value stored in register **I** as each general purpose register is read. This functionality can be enabled by ORing the quirk field with 0x02.