ApyShell/ApyEngine Programming Guide

Version 1.0

Mark Anacker
closecrowd@pm.me
https://www.closecrowd.com

Introduction

ApyEngine is an embeddable interpreter for a subset of the Python 3 language. It runs scripts written in it's version of Python, in a relatively safe, contained way. It can not run anything on it's own, but is designed to be embedded in a host application.

ApyShell is an example of such a host. It can run as a stand-alone program, executing scripts via it's embedded apyengine. It also provides the support structure to manage script files, various directories, and extensions to the engine itself.

And there are the Extension modules. These add controlled, on-demand functionality to the engine to perform specialized tasks. Extensions are written in full Python, and present a uniform interface to the scripts. Many extensions are included with the source distribution, and it's easy to create new ones.

All of this leads to a full-featured scripting environment that can be added to nearly any Python 3 application. It's easy to write scripts for, yet it restricts the abilities of those scripts to cause unexpected or unwanted behavior.

Why use it?

- it's safer than full Python
- it's easier for less-experienced coders to write in
- it's easily merged into an enclosing host application
- It's as powerful as you want to make it, but *only* as powerful as you want

You can grab your own copy at: GitHub . You can download just the apyengine alone, or the full apyshell (which includes apyengine and extensions).

History

The seeds of this project started a few years ago. I had written an Android application for cellular engineers that performed a variety of network tests, and logged the results. It worked well, but adding new test functionality meant re-releasing the app, which became tiresome.

I wanted to give the RF engineers the ability to add new tests, and maybe create their own. That meant a scripting language, easy to code in, and with support for their specific needs. And it had to be safe, so a rogue or defective script could not compromise the test devices.

I could have created a new scripting language, but I picked Python for it's flexibility and vast training resources. Plus, I've been working with Python since the early 1990's, so it's familiar.

The original prototype of the new app was written in Python 2.7, using Kivy as it's UI on Android. It worked, but was pretty primitive compared to the current version.

The project ran it's course, and the app was retired. I took the basic concepts, rewrote the entire codebase from scratch in Python 3, and began using it my own networks. The engine is embedded in my apyshell framework, and has been running system management, Home Automation, and other tasks for a few years now.

Installation

This document assumes that you are using Linux, unless otherwise noted.

Until the setup.py installer is finished, just grab the files from GitHub into a local directory. If you want to leave the defaults unchanged, copy the contents of this directory to “/opt/apyshell”.

The default directories for scripts and extension files are set in apyshell.py:

```
# default directory paths
basedir = '/opt/apyshell/scripts'               # script base directory
extensiondir = '/opt/apyshell/extensions'       # extension base dir
```

You can override these when launching the program with arguments.

These paths are a key piece of the security mechanism. Scripts may only be run from the basedir (or subdirs below it), and only extensions present in the extensiondir directory are available to the scripts.

You can control which scripts may run, and which extensions the scripts can use, by controlling the contents of these directories.

Note: ‘basedir’ must be a single path, which ‘extensiondir’ may be a list of paths, separated by commas.

Running ApyShell

Once you have the files, executing a script is as easy as:

```
./apyshell.py scriptname
```

Assuming that you have created the default directories and moved everything to the right place, **and** have a script named “scriptname.apy”, it’ll start running.

Hint: copy the entire source tree to “/opt/apyshell”, and all the defaults will work out of the box. You can, of course, change the default directories in apyshell.py. And you can set the directories with command-line options. This is useful for running instances of the program with different Extension and script environments.

Built-in help is available for the command-line options:

```
> ./apyshell.py -h

   apyshell.py script [ options ]

   script              The script file to execute (required). The .apy is optional

   -h, --help          This message
   -a, --args          Optional argument string to pass to the script
   -i, --initscript    A script to execute before the specified script
   -b, --basedir       Base directory for scripts (use , for multiple paths)
   -e, --extensiondir  Base directory for extensions (use , for multiple paths)
   -o, --extensionopts A list of options key:value pairs to pass to extensions
   -p, --pidfile       Write a file with the shell's current PID
   -g, --global        All script variables are global
   -v, --verbose       Debug output
```

Note: the options (if any) come **AFTER** the name of the script to execute (and the .apy extension to the script name is optional). If the directory containing apyshell.py is on the path, you can even dispense with the ./ prefix.

To run the included “demo.apy” script at the default locations, just enter:

```
/opt/apyshell/apyshell.py demo
```

Now, say we have a classroom setup where students upload their scripts to a specific directory, and run them against a limited set of extensions. This would be pretty easy to set up with a web server and a bit of CGI. Let’s put the student’s script files in “/class/scripts”, and the curated extensions in “/teacher/apy/extensions”. The program itself can be in /opt/apyshell. To run a script named “prog1”, the command would be:

```
/opt/apyshell/apyshell.py prog1 -b /class/scripts -e /teacher/apy/extensions
```

The scripts in “/class/scripts” can **only** use extensions in “/teacher/apy/extensions”, and can’t wreak havoc on the host system.

Here’s a detailed listing of each option:

```
-h, --help          This message
```

```
-a, --args          Optional argument string to pass to the script
```

You can pass an argument string to your script with this option. For instance, adding -a “myargument” will allow your script to use getSysVar_(“args”) to retrieve it. The value of “args” will be exactly what you pass in “-a”.

```
-i, --initscript    A script to execute before the specified script
```

You can specify a .apy script to execute **before** your script. Since the global state is preserved after the initial script ends, this is an excellent way to initialize variables. Or if you have user-defined functions that are shared among many scripts, this is a way to have them all in one place. When the initial script ends, the main script will be loaded and executed.

```
-b, --basedir       Base directory for scripts (use , for multiple paths)
```

This is the top of the hierarchy of scripts. Apyshell can only execute .apy scripts located in this directory, or in sub-directories below it. You can specify **multiple** base paths by separating them with commas (,). For example:

```
-b “/teacher/commonscripts,/class/scripts”
```

Now both of these paths will be searched for the named .apy files.

```
-e, --extensiondir  Base directory for extensions (use , for multiple paths)
```

This is the directory (or list of directories) where the Extension modules are kept. Only the Extensions in these directories are available to the loadExtension_() command. Like “-b”, separate the paths with commas (,).

```
-o, --extensionopts A list of options key:value pairs to pass to extensions
```

Some Extensions have options controlling various locations or abilities. There are defaults for every Extension, but you can over-ride those defaults with “-o”. For example, the “sqliteext” Extension has two options in apyshell.py:

```
'sql_root':'/opt/apyshell/files'    - The path to use for database files
'sql_ext':'db'                      - The extension to append for db files.
```

If you wanted to change where you stored the Sqlite3 database files, **and** the file extension, you would use something like this:

```
-o ‘sql_root:/opt/apyshell/files,sql_ext:sql’
```

Note that the options are key:value pairs, and multiple options are separated by commas (,). Spaces are okay, no quotes (apart from the enclosing pair).

```
-p, --pidfile       Write a file with the shell's current PID
```

When you run an apyshell script as a background daemon, it’s useful to store it’s Process ID in a file. This makes it easy for application management system to verify that it’s still running. So this option tries to write the PID of the Python process running apyshell to the path and file specified. It doesn’t do any checking of the path – so if you’re running as root, pay attention to the filepath.

```
Example:

-p /var/run/apyshell.pid
```

If the script ends in any sort of controlled manner, apyshell will try to remove the pidfile **if** it was successful in creating it on entry.

```
-g, --global        All script variables are global
```

In standard Python, variables defined in a function are local to that function. If this flag is set, **all** variables everywhere are global. Bad form and all, but useful in some cases. The default is False – variable scope follow the Python norms.

```
-v, --verbose       Debug output
```

Prints some extra debugging info on the console.

Writing .apy Scripts

Now to the good part – writing scripts to run under ApyShell. If you know basic Python, this will look very familiar. The ApyEngine is a Python interpreter – it uses Python's own built-in parser and interpreter to execute your scripts. But there are some differences.

Things not in .apy:

- Classes. While the engine is composed of many Python classes, the .apy scripts do not allow user-defined classes. Maybe in a future version…

- Imports. At least, in the sense of regular Python modules. This is a restricted, controlled environment. Adding new features is done by Extensions, which are carefully written to perform functions in a secure manner. And since we don't have classes, it's be hard to use most Python modules anyways.

An exception to this are the small number of modules available through the "install_()" command. These are Python modules, but with the function names modified to fit the engine's naming conventions. See the sections on Modules later for more information.

- Globals. You can't declare a variable to be global. There are work-arounds, however.

- Lambda, Decorators. Not supported.

- Generators, Yield. Hard to use without classes – not supported.

- Exec, getattr. Nope. There are ways to execute arbitrary commands, if allowed by the local installation.

But look at all the language features that **are** in apy:

- for loops, while loops, if-then-elif-else conditionals
- if-expressions: out = one_thing if TEST else other
- try-except (including 'finally')
- function definitions with def
- advanced slicing: a[::-1], array[-3:, :, ::2]
- arrays, lists, tuples, dicts
- all of the standard Python types
- list comprehension   out = [sqrt(i) for i in values]
- eval (safely in the context of the engine)

In addition, the engine supports extensive thread-safety, since many of the Extension modules are multi-threaded under the hood.

Some conventions

The engine enforces a few naming conventions designed to improve security and readability. It generally follows the Python scheme for naming functions and variables, **except** that your script may not define anything ending with underscore(_). That is reserved for functions installed by the system, Extensions, or internally-created variables.

File names (scripts, data files, etc.) are allowed to use: “a-zA-Z0-9-_”. Directory paths are allowed in some cases (scripts, for instance), but they will be forced to be relative to a host-defined directory.

ApyShell Built-in Functions

Apyshell itself provides several utility functions that you can call. These functions are always available to scripts:

install_(modname)

Installs one of the built-in Python modules so that scripts may use it’s commands.

modname	the module to install. Must be one of: math, time, json, base64, numpy

Returns True if the module installed okay, False if there was a problem.

listModules_()

Returns a list with the names of currently-installed modules.

loadScript_(filename, persist=False)

Executes an .apy script with the specified name, and then continues when it finishes. The script file must be located in one of the directory paths specified with the -b option (or the default path). The “.apy” extension is optional – it’ll be added automatically if not present.

If the optional ‘persist’ argument is True, then any functions defined in the script will be flagged as persistent, and will not be subject to purging later. This is useful for establishing a library of functions that will stick around after their scripts are done.

Here’s an example – three small scripts chained together :

```
# script1.apy
print('script 1 running')
loadScript_('script2')
print('back at script 1')

# script2.apy
print('script 2 running')
loadScript_('script3')
print('back at script 2')

# script3.apy
print('script 3 running')
```

will look like this when run:

```
> apyshell script1
-->  script 1 running
-->  script 2 running
-->  script 3 running
-->  back at script 2
-->  back at script 1
```

The symbol table is global across all these scripts – variables and functions defined in a loaded script will be available to the loading script when it gets control back.

The next four functions deal with symbols (variables and functions) in the environment of the scripts, below the apyengine API line:

```
isDef_( name )
```

Returns True if the symbol named “name” is defined. This can be a variable, or a function.

```
listDefs_()
```

Returns a list of functions that have been defined in the script(s). This is a good way to test if a common function is present before you try to call it.

```
getvar_( vname, default=None )
```

Returns the value of the named script variable, if it exists. Otherwise, return a default value. This is normally called by the **host** application to pass data into the script environment, but it can be called from a script as well. This leads to some interesting meta-programming options.

```
setvar_( vname, val )
```

This function will set the value of a script variable **if**: it already exists, and it’s not set as Read-Only. If the variable exists, and we pass in None for the value, the variable will be deleted from the engine. Normally used by the host application to control a script, but callable from scripts as well.

The next function gives the scripts limited access to data held in the engine, and managed by the host application.

```
getSysVar_( name, default=None )
```

Get a value from the “SysVar” dict, or a default value. SysVar is a dictionary maintained by the engine to pass values between the host side and the script environment. The host application puts useful things in there (such as command-line arguments, hostname, etc.), and the scripts can read them. Scripts can only **get** theses values – the host application can call an engine API function to **set** them.

The next two functions let scripts directly execute the apy dialect of Python:

`check_( code )`

This function does a syntax check of the expression in “code”, but does NOT execute it. Useful it you want to check the expression before you pass it to eval_().

It returns the string “OK” if it’s valid code, “ERR” plus a descriptive message if not. If the “code” argument is empty, returns None.

`eval_( cmd )`

This function executes a Python statement in the context of the engine. It has all of the restrictions of the script that it’s executing in, and shares state with it. This makes it possible to write a REPL, or even a mini-shell within the apy script. You **could** even do something like:

```
    ret = eval_('eval_( "1 + 1" )')
```

but that gets silly.

The return value is the result of the expression (if any), or None if there was no return from the expression. If there was an error, a string will be returned with an error message.

The last two functions end the running of the scripts. A script will normally run until it falls off the end, when it will naturally finish processing. There are two ways to leave in the middle of a running script:

`stop_( ret )`

This function will **gracefully** stop a script and return to apyshell, with an optional integer return code. If this is called in a nested script (i.e. started by loadScript_() in your script), it will exit all the way back out to apyshell.

`exit_( ret )`

This function exits the apyshell interpreter back to the operating system. No cleanup is done – any pidfile is left hanging.

An example

Here’s a small example script showing some of these features. This script (along with may others), is included in the GitHub repository.

demo.apy

```
#
# demo.apy
#
# This little script demonstrates some of the basic
# features of apyshell.
#
# Mark Anacker <closecrowd@pm.me>
# --------------------------------------------------------------------

print('Welcome to apyshell!')

# add the 'time' built-ins
install_('time')

# this gives us the following functions from the Python 'time' module:
#
# ctime_, clock_, asctime_, strptime_, monotonic_,
# gmtime_, mktime_, sleep_, time_, strftime_, localtime_
#
# and constants:
#
# altzone_, timezone_, tzname_, daylight_
#
#


eval_('print("hi")')

# grab the time and date
t = asctime_(localtime_())
print('Right now, it is:', t)

# get our hostname passed in by apyshell:
host = getSysVar_('hostname')
print('We are currently running on '+host)

# and the host Python version
ver = getSysVar_('pythonver')
print('Under Python version:',ver)

#
# Extensions
#

# get the list of extensions that may be loaded by scripts
availlist = scanExtensions_()
print('These extensions are available:', availlist)

# and the list of currently-loaded ones
loadedlist = listExtensions_()
print('These extensions are loaded:', loadedlist)

# load the small utility extension
```

```
loadExtension_('utilext')

# and see the difference
print('Now we can use:', listExtensions_())

# or we can just test for it:
print('Is it loaded:', isExtLoaded_('utilext'))

print('Getting rid of utilext')

# get rid of it...
unloadExtension_('utilext')

# what about now?
print('Is is loaded now:', isExtLoaded_('utilext'))

#
# we can handle exceptions.  cause them, too...
#

print('Exception handling...')

try:
  print('Protected by the try:')
  # fake an error condition
  raise Exception('An error happened')
except Exception as e:
  # note that we give this output line a different prefix...
  print('Except:', str(e), prefix='!!!! ')



print("That's the end for now")
```

When run with “apyshell demo”, this is the output:

```
-->  Welcome to apyshell!
-->  hi
-->  Right now, it is: Mon Nov 27 12:29:01 2023
-->  We are currently running on scrooge
-->  Under Python version: 3.5
-->  These extensions are available: ['mqttext', 'redisext', 'sqliteext', 'utilext',
'fileext', 'queueext']
-->  These extensions are loaded: []
-->  Now we can use: ['utilext']
-->  Is it loaded: True
-->  Getting rid of utilext
-->  Is is loaded now: False
-->  Exception handling...
-->  Protected by the try:
!!!!  Except: An error happened at line 79
-->  That's the end for now
```

Modules

As described previously, certain Python modules may be installed by your scripts. If you don't need the functions of a particular module, you can save some memory by simply not installing it.

The key differences between modules and extensions are:

- modules are pre-determined, and are always available for install.
- once installed, modules remain present until apyshell exits.
- modules add new functions directly into the engine. The new functions don't include parts of the module name (unlike extensions).

The functions added by the modules are direct references to the original Python ones, **except** for the underscore (_) appended to the name. The engine prevents user scripts from defining variables or functions with a trailing _ - if you see that, you know it's a system-supplied symbol.

The Python base module is imported by the install_() command. If the import fails (the module isn't available to the host system), and error message will be printed on the console and the functions won't be available.

There are currently 5 modules available to install: **time**, **json**, **base64**, **math**, and **numpy.** The following sections list each module and the function names supplied by that module..

Time

https://docs.python.org/3/library/time.html

This module provides various time-related functions. It is a direct mapping of most of the Python 3 standard time module, with names slightly changed.

To use:

```
install_(‘time’)
```

Functions available:

```
ctime_(), clock_(), asctime_(), strptime_(), gmtime_(), mktime_(), sleep_(),
time_(), strftime_(), localtime_(), monotonic_()
```

Timezone Constants:

```
altzone_, timezone_ , tzname_ , daylight_
```

Json

https://docs.python.org/3/library/json.html

This module is an interface to the standard Python json module.

To use:

```
install_(‘json’)
```

Functions available:

```
dumps()_,  loads_(),  JSONDecoder_(), JSONEncoder_()
```

Base64

https://docs.python.org/3/library/base64.html

This module provides functions for encoding binary data to printable ASCII characters and decoding such encodings back to binary data.

To use:

```
install_(‘base64’)
```

Functions available:

```
b64encode_(), b64decode_(), urlsafe_b64encode_(), urlsafe_b64decode_()
```

Math

https://docs.python.org/3/library/math.html

This module provides access to the mathematical functions defined by the C standard. It provides all of the math functions as of Python 3.5. These calls are equivalent to the Python version, without the module prefix. For example: “cos_()” is the same as “math.cos()” - apyengine doesn’t support namespaces.

To use:

```
install_(‘math’)
```

Functions available:

```
acos_(), acosh_(), asin_(), asinh_(), atan_(), atan2_(), atanh_(), ceil_(),
copysign_(), cos_(), cosh_(), degrees_(), exp_(), fabs_(), factorial_(), floor_(),
fmod_(), frexp_(), fsum_(), hypot_(), isinf_(), isnan_(), ldexp_(), log_(),
log10_(), log1p_(), modf_(), pow_(), radians_(), sin_(), sinh_(), sqrt_(), tan_(),
tanh_(), trunc_()
```

Pre-defined constants:

```
e_, pi_, inf_, nan_
```

Numpy

https://numpy.org/

If you have the numpy module installed on your host system, this modules will allow your script to use many of the functions in this extensive package. Familiarity with Numpy is very much encouraged.

To use:

```
install_('numpy')
```

Functions available:

```
Inf_(), NAN_(), abs_(), add_(), alen_(), all_(), amax_(), amin_(), angle_(),
any_(), append_(), arange_(), arccos_(), arccosh_(), arcsin_(), arcsinh_(),
arctan_(), arctan2_(), arctanh_(), argmax_(), argmin_(), argsort_(), argwhere_(),
around_(), array_(), array2string_(), asanyarray_(), asarray_(),
asarray_chkfinite_(), ascontiguousarray_(), asfarray_(), asfortranarray_(),
asmatrix_(), asscalar_(), atleast_1d_(), atleast_2d_(), atleast_3d_(), average_(),
bartlett_(), base_repr_(), bitwise_and_(), bitwise_not_(), bitwise_or_(),
bitwise_xor_(), blackman_(), bool_(), broadcast_(), broadcast_arrays_(), byte_(),
c__(), cdouble_(), ceil_(), cfloat_(), chararray_(), choose_(), clip_(),
clongdouble_(), clongfloat_(), column_stack_(), common_type_(), complex_(),
complex128_(), complex64_(), complex__(), complexfloating_(), compress_(),
concatenate_(), conjugate_(), convolve_(), copy_(), copysign_(), corrcoef_(),
correlate_(), cos_(), cosh_(), cov_(), cross_(), csingle_(), cumprod_(), cumsum_(),
datetime_data_(), deg2rad_(), degrees_(), delete_(), diag_(), diag_indices_(),
diag_indices_from_(), diagflat_(), diagonal_(), diff_(), digitize_(), divide_(),
dot_(), double_(), dsplit_(), dstack_(), dtype_(), e_(), ediff1d_(), empty_(),
empty_like_(), equal_(), exp_(), exp2_(), expand_dims_(), expm1_(), extract_(),
eye_(), fabs_(), fill_diagonal_(), finfo_(), fix_(), flatiter_(), flatnonzero_(),
fliplr_(), flipud_(), float_(), float32_(), float64_(), float__(), floating_(),
floor_(), floor_divide_(), fmax_(), fmin_(), fmod_(), format_parser_(), frexp_(),
frombuffer_(), fromfile_(), fromfunction_(), fromiter_(), frompyfunc_(),
fromregex_(), fromstring_(), fv_(), genfromtxt_(), getbufsize_(), geterr_(),
gradient_(), greater_(), greater_equal_(), hamming_(), hanning_(), histogram_(),
histogram2d_(), histogramdd_(), hsplit_(), hstack_(), hypot_(), i0_(), identity_(),
iinfo_(), imag_(), in1d_(), index_exp_(), indices_(), inexact_(), inf_(), info_(),
infty_(), inner_(), insert_(), int_(), int0_(), int16_(), int32_(), int64_(),
int8_(), int__(), int_asbuffer_(), intc_(), integer_(), interp_(), intersect1d_(),
intp_(), invert_(), ipmt_(), irr_(), iscomplex_(), iscomplexobj_(), isfinite_(),
isfortran_(), isinf_(), isnan_(), isneginf_(), isposinf_(), isreal_(),
isrealobj_(), isscalar_(), issctype_(), iterable_(), ix__(), kaiser_(), kron_(),
ldexp_(), left_shift_(), less_(), less_equal_(), linspace_(), little_endian_(),
load_(), loads_(), loadtxt_(), log_(), log10_(), log1p_(), log2_(), logaddexp_(),
logaddexp2_(), logical_and_(), logical_not_(), logical_or_(), logical_xor_(),
logspace_(), long_(), longcomplex_(), longdouble_(), longfloat_(), longlong_(),
mafromtxt_(), mask_indices_(), mat_(), matrix_(), maximum_(), maximum_sctype_(),
may_share_memory_(), mean_(), median_(), memmap_(), meshgrid_(), mgrid_(),
minimum_(), mintypecode_(), mirr_(), mod_(), modf_(), msort_(), multiply_(),
nan_(), nan_to_num_(), nanargmax_(), nanargmin_(), nanmax_(), nanmin_(), nansum_(),
ndarray_(), ndenumerate_(), ndfromtxt_(), ndim_(), ndindex_(), negative_(),
newaxis_(), nextafter_(), nonzero_(), not_equal_(), nper_(), npv_(), number_(),
obj2sctype_(), ogrid_(), ones_(), ones_like_(), outer_(), packbits_(),
percentile_(), pi_(), piecewise_(), place_(), pmt_(), poly_(), poly1d_(),
polyadd_(), polyder_(), polydiv_(), polyfit_(), polyint_(), polymul_(), polysub_(),
polyval_(), power_(), ppmt_(), prod_(), product_(), ptp_(), put_(), putmask_(),
pv_(), r__(), rad2deg_(), radians_(), rank_(), rate_(), ravel_(), real_(),
real_if_close_(), reciprocal_(), record_(), remainder_(), repeat_(), reshape_(),
resize_(), restoredot_(), right_shift_(), rint_(), roll_(), rollaxis_(), roots_(),
rot90_(), round_(), round__(), row_stack_(), s__(), sctype2char_(),
searchsorted_(), select_(), setbufsize_(), setdiff1d_(), seterr_(), setxor1d_(),
shape_(), short_(), sign_(), signbit_(), signedinteger_(), sin_(), sinc_(),
single_(), singlecomplex_(), sinh_(), size_(), sometrue_(), sort_(),
sort_complex_(), spacing_(), split_(), sqrt_(), square_(), squeeze_(), std_(),
str_(), str__(), subtract_(), sum_(), swapaxes_(), take_(), tan_(), tanh_(),
tensordot_(), tile_(), trace_(), transpose_(), trapz_(), tri_(), tril_(),
tril_indices_(), tril_indices_from_(), trim_zeros_(), triu_(), triu_indices_(),
triu_indices_from_(), true_divide_(), trunc_(), ubyte_(), uint_(), uint0_(),
uint16_(), uint32_(), uint64_(), uint8_(), uintc_(), uintp_(), ulonglong_(),
union1d_(), unique_(), unravel_index_(), unsignedinteger_(), unwrap_(), ushort_(),
vander_(), var_(), vdot_(), vectorize_(), vsplit_(), vstack_(), where_(), who_(),
zeros_(), zeros_like_(), fft_(), linalg_(), polynomial_(), random_()
```

A full discussion of Numpy is out of the scope of this document.

Extensions

Extensions are optional add-ons that perform complex tasks in the background, while presenting a simplified, consistent interface to the scripts. This is where the real power of the apyengine comes through. Instead of having to know the gory details of all the Python modules a coder might want to use, they can just use the extensions. The messy bits are hidden under a clean interface.

Unlike the Modules, Extensions may or may not be available to a set of scripts. When apyshell is started, it may be passed an option to use a specific extension directory list (-e). Only the extensions in those directories are available to be loaded. Thus you can control what the scripts running under a given invocation of apyshell can do.

And also unlike Modules, Extensions may be unloaded. If you are finished with an Extension, and you want the resources back, simply call the "unloadExtension_()" function with the name of the Extension.

Some Extensions, especially those that bring in data from the outside, have the option of a "handler". This is simply a function, defined in your script, that gets called whenever data is available for processing. So instead of continuously polling a function, you can specify the handler and go about your other business. The handler will be called when data is ready.

The handlers generally run in the context of a thread running in the Extension. So it's best to do as little as possible in the handler itself, and then put the data into a Queue for the main code to process when it can. This sort of event-driven architecture can be very flexible, and fast. Of course, you can just poll in a loop for your data also.

The following is a list of the currently-supplied Extensions. They are fully commented, and may be used as templates for adding new ones.

fileext
scripts/fileext.py

This extension provides some simple text file handling. It limits the file locations to a pre-set directory and below (the “file_root” option).

To install:

```
loadExtension_(‘fileext’)
```

To uninstall:

```
unloadExtension_(‘fileext’)
```

This extension provides the following functions. Note that some are optional:

```
readLines_(filepath, handler=None, maxlines=0)
```

Reads lines from a text file (all or up to a specified count) . If the “handler” argument is present, each line (up to maxlines) is passed to the handler as it’s read. Otherwise, the text is accumulated in a buffer and passed back when it hits maxlines or EOF.

```
writeLines_(filepath, data, handler=None, maxlines=0)
```

Writes lines to a text file (all or up to a specified count) . If the “handler” argument is present, the handler functions is called for a line of text (up to maxlines), which is then written to the file.. Otherwise, the text in the data buffer is written line-by-line until it hits maxlines or the end of the buffer.

If the file exists, it’s overwritten.

```
appendLines_(filepath, data, handler=None, maxlines=0)
```

Writes lines to a text file (all or up to a specified count) . If the “handler” argument is present, the handler functions is called for a line of text (up to maxlines), which is then written to the file.. Otherwise, the text in the data buffer is written line-by-line until it hits maxlines or the end of the buffer.

If the file doesn’t exists, it’s created. Otherwise, lines are appended to the end of the file.

```
listFiles_(filepath=’’)
```

Returns a list of files and subdirectories, either in the file_root, or a path **below** it.

There are some extension options defined in apyshell.py that modify this Extension:

'file_root':'/opt/apyshell/files'	All file activity through this Extension happens in or under this path.
'read_only':False	If True, only the readLines_() and listFiles_() functions are installed.
'list_files':True	If False, the listFiles_() function is not installed.

Example:

Here’s a simple sample script that doesn’t use a handler:

```
r = loadExtension_('fileext')
l = listFiles_()
print(‘Files:’, l)
n = readLines_('w.txt')
print(n)
```

Assuming that there’s a single file in the file_root directory named “w.txt”, this is what the output would look like:

```
-->  Files: ['w.txt']
-->  line1
line2
line3
```

And here’s a quick one with a handler:

```
r = loadExtension_('fileext')

# this is a handler - it gets each line as it
# is read.
def oneline(l):
    print(l, prefix='->->')

# pass each line through the handler
n = readLines_('w.txt', handler='oneline')
```

Produces:

```
->-> line1
->-> line2
->-> line3
```

utilext
utilext.py

This extension provides an assortment of useful functions that don’t fit in elsewhere.

To install:

```
loadExtension_(‘utilext’)
```

To uninstall:

```
unloadExtension_(‘utilext’)
```

This extension provides the following functions. Note that some are optional:

```
input_(prompt=None, default=None, **kwargs)
```

This function will (optionally) print a prompt on the console, then wait for (and return) whatever the user types in. A default value may be set to be returned if nothing was entered. A timeout can also be set, either returning a default value or raising an exception if the timeout expires without a console entry.

The options that may be added (kwargs) are:

timeout	:	Input timeout in seconds. The default is 0 (no timeout).
todef	:	Default value to return if the timeout expires.
toraise	:	If True, raise an Exception when the timeout expires instead of returning the default value.

```
getenv_(name)
```

This function returns the value of an item in the host application's environment. Since this might be unsafe, it has to be specifically enabled by the host application in the extension options.

```
system_(cmd)
```

This function simply runs whatever string is passed to it in the host environment. Useful for debugging or calling external applications, it has to be deliberately enabled by the host application.   <u>This is rather dangerous</u> – you should probably set the default in apyshell.py to False unless you really need it.

There are some extension options defined in apyshell.py that modify this Extension:

'allow_system':True	Install the system_() function
'allow_getenv':True	Install the getenv_() function

Example:

demoinput.apy

```
# load the extension
loadExtension_('utilext')

# prompt the user to enter something.  If they don't within 5 seconds,
# the string "time!" will be be returned
try:
	f = input_('type something ', '--', timeout=5, todef='time!', toraise=False)
	print('you typed:', f)
except Exception as e:
	print('exception:', str(e), prefix=None)
```

Here we enter a line on the console:

```
type something okay
-->  you typed: okay
```

`And the text we typed was returned.` Here we don’t (the toraise option is False):

```
type something -->  you typed: time!
```

The default value was returned. When toraise is True and the timeout expires:

```
type something exception: Error running function call 'input_' with args ['type something ',
'--'] and kwargs {'timeout': 5,
'toraise': True, 'todef': 'time!'}: Timed out at line 16
```

Like another well-known scripting language, there’s more than one way to do something.

queueext
qeueuext.py

This extension provides thread-safe queues for communicating between threads (and callback handlers). It has all the queue management functions anyone should need.

Queues are referenced by name (set in the queue_open_() function), and may be either a classic First-In-First-Out queue, or a Last-In-First-Out stack. These queues are an excellent way to pass data from an event-driven handler function to a main processing loop.

To install:

loadExtension_('utilext')

To uninstall:

unloadExtension_('utilext')

This extension provides the following functions:

queue_open_(name, **kwargs)

Creates a queue referenced by "name" (if there isn't already a queue with that name). This queue will initially be empty.

The option that may be added (kwargs) is:

type : Either 'fifo' for a regular queue, or 'lifo' for a stack. The default is 'fifo'.

It will return True if the queue was created, False if there was an error.

queue_close_(name)

Empties the named queue (if it exists), and deletes it.   Any script elements (handlers, tasks, etc.) that are waiting on a queue_get_() will return immediately.

It will return True if the queue was removed, False if there was an error.

queue_put_(name, value, **kwargs)

Adds the given value to the queue (tail (fifo) or head (lifo)). If value is None, this function simply returns False.

The options that may be added (kwargs) are:

block : If True, block until there is room in the queue to add the value, **or** the timeout expires, **or** the queue is closed.

timeout: The number of seconds to wait for a blocking put to succeed before returning False. The default is 0, which means block forever (or the queue is closed).

It will return True if the value was added to the queue, False if there was an error.

queue_get_(name, **kwargs)

Gets an item from the queue. A 'fifo' queue returns the item at the HEAD of the queue. A 'lifo' type will return the item at the END of the queue.

The get operation may be either blocking or non-blocking. If the block option in kwargs is False, the operation will return immediately. If the queue was empty, the return will be None.

If block is True (default) and the queue is empty, the get operation will be retried at 1-second intervals until either: an item is added to the queue, the timeout count is reached, or the queue is closed. A timeout of 0 will retry until either success or close.

The options that may be added (kwargs) are:

block : If True, block until there is an item in the queue to return, **or** the timeout expires, **or** the queue is closed.

timeout: The number of seconds to wait for a blocking get to succeed before returning False. The default is 0, which means block forever (or the queue is closed).

It will return the next item from the queue, or None if: - the queue is empty and block==False, - block == True and the timeout expired, or – the queue was closed().

queue_clear_(name)

Removes all the items in a queue.

It will return True if the queue was cleared, False if there was an error.

queue_len_(name)

Returns the number of items in the named queue.

It will return 0 if there was an error.

queue_isempty_(name)

Returns True if the queue is empty.

queue_list_()

Returns a list of the currently-open queues. If there are no queues, it returns an empty list.

It will return None if there was an error.