EnergyPlus Python API

The Python bindings around the C API are documented here. The API is categorized into the following sections, with the functionality at the root of the API documented below.

Contents:

class **`api.EnergyPlusAPI`**(*running_as_python_plugin: bool = False*) [source]

Bases: `object`

This class exposes the EnergyPlus C Library API to Python. The API is split into three categories, and this class exposes each API category through member variables. If an instance of this class is created as *api = EnergyPlusAPI()*, then the following members are available:

- *api.functional*: The functional API provides access to static API calls, such as thermophysical property methods.
- *api.runtime*: The runtime API allows a user to provide Python functions as callbacks, which are then called from within EnergyPlus at specific points in the simulation.
- *api.exchange* The data_exchange API allows a user to exchange data (get sensor values, set actuator values) from within runtime callback methods, during a simulation. When this class is instantiated for Python Plugin use, this also exposes the plugin global data members to allow sharing data between plugins.

In a makefile-style build, the API library (dll) should be in Products; for example: */path_to_build/Products*. For Visual Studio, the DLL is inside of a Debug or Release folder inside that Products directory. At build time, the cmake/PythonSetupAPIinBuild.cmake script is executed (the energyplusapi target depends on it). At build time, the Python API files are placed inside of the Products directory on Makefile builds, and copied into *both* the Release and Debug folders on Visual Studio builds. The API scripts are put into a pyenergyplus directory, so in all cases, the dynamic library will simply be in the current script's parent directory. In an installation, the library will be in the installation root, and the Python API files will be in a pyenergyplus directory inside that install root as well, so the binary will again just be in this script's parent directory.

For either case, utilizing the Python API wrappers is straightforward: if executing from directly from the buil
folder, scripts can be imported as *from pyenergyplus.foo import bar*. If executing from a totally separate direc
install directory can be inserted into the beginning of sys.path so that the pyenergyplus directory can be found.

stable

To reference this in an IDE to allow writing scripts using autocomplete, etc., most IDEs allow you to add third- party library directories. The directory to add would be the build or install folder, as appropriate, so that the *from pyenergyplus* import statements can find a pyenergyplus package inside that third-party directory.

static `api_version`()→ str [source]

Returns a string representation of the version of this API. The EnergyPlus API will evolve over time, but in most cases, it will be simply adding new functionality and methods, not breaking existing API calls. The fractional portion of the API will be incremented when new functionality is added, and the whole number portion will be incremented when an existing API is broken. :return:

`verify_api_version_match`(*state: c_void_p*)→ None [source]

`api.api_path`()→ str [source]

This function returns a string to the EnergyPlus dynamic library. The energyplusapi target in the build system depends on the Python API build script, so you shouldn't be able to generate the EnergyPlus dynamic library without these scripts being successfully set up in the build tree by CMake.

Returns: A string absolute path to the EnergyPlus DLL.

State API

What is this *state* business?

OK, so you are using this API and may have seen *state* as an argument floating around the API functions (or inside the code if you looked close enough). Up until recently, it was standard practice in EnergyPlus to put all the data in the global state of the program. Because the program was always run one way - a single exe thread - this was not a problem because the data was deleted when the program ended.

EnergyPlus is venturing into diverse applications and workflows, including running as a library, as a service, embedded in controllers, used with hardware in the loop and other research experimentation For these applications, having a huge blob of unmanaged global data is a problematic design for multiple reasons:

1. The data in the program is not cleared when you call EnergyPlus as a library multiple times in the same thread. The second time a run is started, previous simulation data will be present if the data is not cleared. While this does not explicitly require moving away from global data, having an unmanaged blob of thousands of global variables to reset is a maintenance nightmare.
2. For some of those applications, it's not just about clearing the global state, but also being able to save and reload the state of the program in order to be used as part of control decisions will be important.
3. It is impossible for EnergyPlus to run multi-threaded in the same process, because the global variables are not thread-safe (and adding thread-local to thousands of global variables is a no-go for performance reasons). With many new workflows, on machines that have plenty of CPU cores available, running EnergyPlus as part of a multi-threaded library will be heavily beneficial.

So what does this have to do with this new *state* variable I've seen around? Think of this as the instance of EnergyPlus that you are currently running. In a fully OO design, this could just be the object instance that you create and then execute methods on that instance. EnergyPlus is quickly moving away from global state, and into "managed" state, but some global data still remains in the program. This state object holds a reference to the ever growing "managed" state, which will eventually encapsulate the entire running state of the program.

From an API client perspective, nothing ever needs to be manually done with this object. Simply create the instance of state and pass it into all the API functions and let EnergyPlus do what it needs to do. API callback functions receive a payload of the running state instance so that it can again be passed into API functions within the callback. Python plugin override methods also accept a state instance argument so that API function calls can be made with the running state. State really is just a small payload that needs to be properly passed around through the API.

class `state.StateManager`(*api: <ctypes.LibraryLoader object at 0x769a47078950>*) [source]

Bases: `object`

This API class enables a client to create and manage state instances for using EnergyPlus API methods. Nearly all EnergyPlus API methods require a state object to be passed in, and when callbacks are made, the current state is passed as the only argument. This allows client code to close the loop and pass the current state when making API calls inside callbacks.

The state object is at the heart of accessing EnergyPlus via API, however, the client code should simply be a courier of this object, and never attempt to manipulate the object. State manipulation occurs inside EnergyPlus, and attempting to modify it manually will likely not end well for the workflow.

This class allows a client to create a new state, reset it, and free the object when finished with it.

stable

`delete_state`(*state: c_void_p*)→ None [source]

This function deletes an existing state instance, freeing the memory.

Returns: Nothing

`new_state()`→ c_void_p [source]

This function creates a new state object that is required to pass into EnergyPlus Runtime API function calls

Returns: A pointer to a new state object in memory

`reset_state`(*state: c_void_p*)→ None [source]

This function resets an existing state instance, thus resetting the simulation, including any registered callback functions.

Returns: Nothing

Functional API

class `func.EnergyPlusVersion` [source]

Bases: `object`

This is the EnergyPlus version. Could also call into the DLL but it’s the same effect.

class `func.Functional`(*api: <ctypes.LibraryLoader object at 0x769a47078950>, running_as_python_plugin: bool = False*) [source]

Bases: `object`

This API class enables accessing structures and functionality inside EnergyPlus from an outside client. This functional API will be extended over time, but initial targeted functionality includes fluid and refrigerant property methods, and surface and geometry classes and methods.

The Functional API class itself is really just an organizational class that provides access to nested functional classes through member functions. The functional API class is instantiated by the higher level EnergyPlusAPI class, and clients should *never* attempt to create an instance manually. Instead, create an EnergyPlusAPI instance, and use the *functional* member variable to access a Functional class instance. For Python Plugin workflows, the EnergyPlusPlugin base class also provides an instance of the Functional base class through the *self.api.functional* member variable. Clients should use that directly when needing to make functional calls into the library.

`callback_error`(*state, f: LambdaType*)→ None [source]

This function allows a client to register a function to be called back by EnergyPlus when an error message is added to the error file. The user can then detect specific error messages or whatever.

Parameters:
- **state** – An active EnergyPlus “state” that is returned from a call to *api.state_manager.new_state()*.
- **f** – A python function which takes an integer severity and a string (bytes) argument and returns nothing

Returns: Nothing

static `clear_callbacks`()→ None [source]

This function is only used if you are running this script continually making many calls into the E+ library in one thread, each with many new and different error handler callbacks, and you need to clean up.

Note this will affect all current instances in this thread, so use carefully!

Returns: Nothing

static `ep_version`()→ EnergyPlusVersion [source]

`glycol`(*state: c_void_p, glycol_name: str*)→ Glycol [source]

Returns a Glycol instance, which allows calculation of glycol properties.

Parameters:
- **state** – An active EnergyPlus “state” that is returned from a call to *api.state_manager.new_state()*.
- **glycol_name** – Name of the Glycol, for now only water is allowed

Returns: An instantiated Glycol structure

`initialize`(*state: c_void_p*)→ None [source]

stable

`psychrometrics`(*state: c_void_p*)→ Psychrometrics [source]

Returns a Psychrometric instance, which allows calculation of psychrometric properties.

Parameters: **state** – An active EnergyPlus “state” that is returned from a call to *api.state_manager.new_state()*.
Returns: An instantiated Psychrometric structure

`refrigerant`(*state: c_void_p*, *refrigerant_name: str*)→ Refrigerant [source]

Returns a Refrigerant instance, which allows calculation of refrigerant properties.

Parameters:
- **state** – An active EnergyPlus “state” that is returned from a call to *api.state_manager.new_state()*.
- **refrigerant_name** – Name of the Refrigerant, for now only steam is allowed

Returns: An instantiated Refrigerant structure

class `func.Glycol`(*state: ~ctypes.c_void_p*, *api: <ctypes.LibraryLoader object at 0x769a47078950>*, *glycol_name: bytes*) [source]

Bases: `object`

This class provides access to the glycol property calculations inside EnergyPlus. For now, the only glycol name allowed is plain water. This is because other fluids are only initialized when they are declared in the input file. When calling through the API, there is no input file, so no other fluids are declared. This is ripe for a refactor to enable additional fluids, but water will suffice for now.

`conductivity`(*state: c_void_p*, *temperature: float*)→ float [source]

Returns the conductivity of the fluid at the specified temperature.

Parameters:
- **state** – An active EnergyPlus “state” that is returned from a call to *api.state_manager.new_state()*
- **temperature** – Fluid temperature, in degrees Celsius

Returns: The conductivity of the fluid, in W/m-K

`delete`(*state: c_void_p*)→ None [source]

Frees the memory of the associated Glycol instance inside the EnergyPlus (C++) state.

Parameters: **state** – An active EnergyPlus “state” that is returned from a call to *api.state_manager.new_state()*.
Returns: Nothing

`density`(*state: c_void_p*, *temperature: float*)→ float [source]

Returns the density of the fluid at the specified temperature.

Parameters:
- **state** – An active EnergyPlus “state” that is returned from a call to *api.state_manager.new_state()*
- **temperature** – Fluid temperature, in degrees Celsius

Returns: The density of the fluid, in kg/m3

`specific_heat`(*state: c_void_p*, *temperature: float*)→ float [source]

Returns the specific heat of the fluid at the specified temperature.

Parameters:
- **state** – An active EnergyPlus “state” that is returned from a call to *api.state_manager.new_state()*
- **temperature** – Fluid temperature, in degrees Celsius

Returns: The specific heat of the fluid, in J/kg-K

`viscosity`(*state: c_void_p*, *temperature: float*)→ float [source]

Returns the dynamic viscosity of the fluid at the specified temperature.

Parameters:
- **state** – An active EnergyPlus “state” that is returned from a call to *api.state_manager.new_state()*
- **temperature** – Fluid temperature, in degrees Celsius

Returns: The dynamic viscosity of the fluid, in Pa-s (or kg/m-s) *class* **func.Psychrometrics**(*api: <ctypes.LibraryLoader object at 0x769a47078950>*) [source]

Bases: `object`

This class provides access to the psychrometric functions within EnergyPlus. Some property calculations are available as functions of different independent variable combinations, leading to suffixed function names, such as *vapor_density_b* and *relative_humidity_c*.

density(*state: c_void_p, barometric_pressure: float, dry_bulb_temp: float, humidity_ratio: float*)→ float [source]

Returns the psychrometric density at the specified conditions.

Parameters:
- **state** – An active EnergyPlus “state” that is returned from a call to *api.state_manager.new_state()*.
- **barometric_pressure** – Barometric pressure, in Pa
- **dry_bulb_temp** – Psychrometric dry bulb temperature, in C
- **humidity_ratio** – Humidity ratio, in kgWater/kgDryAir

Returns:

dew_point(*state: c_void_p, humidity_ratio: float, barometric_pressure: float*)→ float [source]

Returns the psychrometric dew point temperature at the specified conditions.

Parameters:
- **state** – An active EnergyPlus “state” that is returned from a call to *api.state_manager.new_state()*.
- **humidity_ratio** – Humidity ratio, in kgWater/kgDryAir
- **barometric_pressure** – Barometric pressure, in Pa

Returns:

dew_point_b(*state: c_void_p, dry_bulb_temp: float, wet_bulb_temp: float, barometric_pressure: float*)→ float [source]

Returns the psychrometric dew point temperature at the specified conditions.

Parameters:
- **state** – An active EnergyPlus “state” that is returned from a call to *api.state_manager.new_state()*.
- **dry_bulb_temp** – Psychrometric dry bulb temperature, in C
- **wet_bulb_temp** – Psychrometric wet bulb temperature, in C
- **barometric_pressure** – Barometric pressure, in Pa

Returns:

dry_bulb(*state: c_void_p, enthalpy: float, humidity_ratio: float*)→ float [source]

Returns the psychrometric dry bulb temperature at the specified conditions.

Parameters:
- **state** – An active EnergyPlus “state” that is returned from a call to *api.state_manager.new_state()*.
- **enthalpy** – Psychrometric enthalpy, in J/kg
- **humidity_ratio** – Humidity ratio, in kgWater/kgDryAir

Returns:

enthalpy(*state: c_void_p, dry_bulb_temp: float, humidity_ratio: float*)→ float [source]

Returns the psychrometric enthalpy at the specified conditions.

Parameters:
- **state** – An active EnergyPlus “state” that is returned from a call to *api.state_manager.new_state()*.
- **dry_bulb_temp** – Psychrometric dry bulb temperature, in C
- **humidity_ratio** – Humidity ratio, in kgWater/kgDryAir

Returns:

enthalpy_b(*state: c_void_p, dry_bulb_temp: float, relative_humidity_fraction: float, barometric_pressure: float*)→ float [source]

Returns the psychrometric enthalpy at the specified conditions.

stable

Parameters:
- **state** – An active EnergyPlus “state” that is returned from a call to *api.state_manag*
- **dry_bulb_temp** – Psychrometric dry bulb temperature, in C

- **relative_humidity_fraction** – Psychrometric relative humidity, as a fraction from 0.0 to 1.0.
- **barometric_pressure** – Barometric pressure, in Pa

Returns:

`humidity_ratio`(*state: c_void_p*, *dry_bulb_temp: float*, *enthalpy: float*)→ float [source]

Returns the psychrometric humidity ratio at the specified conditions.

Parameters:
- **state** – An active EnergyPlus “state” that is returned from a call to *api.state_manager.new_state()*.
- **dry_bulb_temp** – Psychrometric dry bulb temperature, in C
- **enthalpy** – Psychrometric enthalpy, in J/kg

Returns:

`humidity_ratio_b`(*state: c_void_p*, *dew_point_temp: float*, *barometric_pressure: float*)→ float [source]

Returns the psychrometric humidity ratio at the specified conditions.

Parameters:
- **state** – An active EnergyPlus “state” that is returned from a call to *api.state_manager.new_state()*.
- **dew_point_temp** – Psychrometric dew point temperature, in Celsius
- **barometric_pressure** – Barometric pressure, in Pa

Returns:

`humidity_ratio_c`(*state: c_void_p*, *dry_bulb_temp: float*, *relative_humidity_fraction: float*, *barometric_pressure: float*)→ float [source]

Returns the psychrometric humidity ratio at the specified conditions.

Parameters:
- **state** – An active EnergyPlus “state” that is returned from a call to *api.state_manager.new_state()*.
- **dry_bulb_temp** – Psychrometric dry bulb temperature, in C
- **relative_humidity_fraction** – Psychrometric relative humidity, as a fraction from 0.0 to 1.0.
- **barometric_pressure** – Barometric pressure, in Pa

Returns:

`humidity_ratio_d`(*state: c_void_p*, *dry_bulb_temp: float*, *wet_bulb_temp: float*, *barometric_pressure: float*)→ float [source]

Returns the psychrometric humidity ratio at the specified conditions.

Parameters:
- **state** – An active EnergyPlus “state” that is returned from a call to *api.state_manager.new_state()*.
- **dry_bulb_temp** – Psychrometric dry bulb temperature, in C
- **wet_bulb_temp** – Psychrometric wet bulb temperature, in C
- **barometric_pressure** – Barometric pressure, in Pa

Returns:

`latent_energy_of_air`(*state: c_void_p*, *dry_bulb_temp: float*)→ float [source]

Returns the psychrometric latent energy of air at the specified conditions.

Parameters:
- **state** – An active EnergyPlus “state” that is returned from a call to *api.state_manager.new_state()*.
- **dry_bulb_temp** – Psychrometric dry bulb temperature, in C

Returns:

`latent_energy_of_moisture_in_air`(*state: c_void_p*, *dry_bulb_temp: float*)→ float [source]

Returns the psychrometric latent energy of the moisture in air at the specified conditions.

Parameters:
- **state** – An active EnergyPlus “state” that is returned from a call to *api.state_manager.new_state()*.
- **dry_bulb_temp** – Psychrometric dry bulb temperature, in C

Returns:

stable

`relative_humidity`(*state: c_void_p*, *dry_bulb_temp: float*, *vapor_density: float*)→ float [source]

Returns the psychrometric relative humidity at the specified conditions.

Parameters:
- **state** – An active EnergyPlus “state” that is returned from a call to *api.state_manager.new_state()*.
- **dry_bulb_temp** – Psychrometric dry bulb temperature, in C
- **vapor_density** – Psychrometric vapor density, in kg/m3

Returns:

`relative_humidity_b`(*state: c_void_p*, *dry_bulb_temp: float*, *humidity_ratio: float*, *barometric_pressure: float*)→ float [source]

Returns the psychrometric relative humidity at the specified conditions.

Parameters:
- **state** – An active EnergyPlus “state” that is returned from a call to *api.state_manager.new_state()*.
- **dry_bulb_temp** – Psychrometric dry bulb temperature, in C
- **humidity_ratio** – Humidity ratio, in kgWater/kgDryAir
- **barometric_pressure** – Barometric pressure, in Pa

Returns:

`saturation_pressure`(*state: c_void_p*, *dry_bulb_temp: float*)→ float [source]

Returns the psychrometric saturation pressure at the specified conditions.

Parameters:
- **state** – An active EnergyPlus “state” that is returned from a call to *api.state_manager.new_state()*.
- **dry_bulb_temp** – Psychrometric dry bulb temperature, in C

Returns:

`saturation_temperature`(*state: c_void_p*, *enthalpy: float*, *barometric_pressure: float*)→ float [source]

Returns the psychrometric saturation temperature at the specified conditions.

Parameters:
- **state** – An active EnergyPlus “state” that is returned from a call to *api.state_manager.new_state()*.
- **enthalpy** – Psychrometric enthalpy, in J/kg
- **barometric_pressure** – Barometric pressure, in Pa

Returns:

`specific_heat`(*state: c_void_p*, *humidity_ratio: float*)→ float [source]

Returns the psychrometric specific heat at the specified conditions.

Parameters:
- **state** – An active EnergyPlus “state” that is returned from a call to *api.state_manager.new_state()*.
- **humidity_ratio** – Humidity ratio, in kgWater/kgDryAir

Returns:

`specific_volume`(*state: c_void_p*, *dry_bulb_temp: float*, *humidity_ratio: float*, *barometric_pressure: float*)→ float [source]

Returns the psychrometric specific volume at the specified conditions.

Parameters:
- **state** – An active EnergyPlus “state” that is returned from a call to *api.state_manager.new_state()*.
- **dry_bulb_temp** – Psychrometric dry bulb temperature, in C
- **humidity_ratio** – Humidity ratio, in kgWater/kgDryAir
- **barometric_pressure** – Barometric pressure, in Pa

Returns:

`vapor_density`(*state: c_void_p*, *dry_bulb_temp: float*, *humidity_ratio: float*, *barometric_pressure: float*)→ float [source]

Returns the psychrometric vapor density at the specified conditions.

Parameters:
- **state** – An active EnergyPlus “state” that is returned from a call to *api.state_manager.new_state()*.
- **dry_bulb_temp** – Psychrometric dry bulb temperature, in C
- **humidity_ratio** – Humidity ratio, in kgWater/kgDryAir
- **barometric_pressure** – Barometric pressure, in Pa

stable

Returns:

`vapor_density_b`(*state: c_void_p*, *dry_bulb_temp: float*, *relative_humidity_fraction: float*)→ float [source]

Returns the psychrometric vapor density at the specified conditions.

Parameters:
- **state** – An active EnergyPlus “state” that is returned from a call to *api.state_manager.new_state()*.
- **dry_bulb_temp** – Psychrometric dry bulb temperature, in C
- **relative_humidity_fraction** – Psychrometric relative humidity, as a fraction from 0.0 to 1.0.

Returns:

`wet_bulb`(*state: c_void_p*, *dry_bulb_temp: float*, *humidity_ratio: float*, *barometric_pressure: float*)→ float [source]

Returns the psychrometric wet bulb temperature at the specified conditions.

Parameters:
- **state** – An active EnergyPlus “state” that is returned from a call to *api.state_manager.new_state()*.
- **dry_bulb_temp** – Psychrometric dry bulb temperature, in C
- **humidity_ratio** – Humidity ratio, in kgWater/kgDryAir
- **barometric_pressure** – Barometric pressure, in Pa

Returns:

class **`func.Refrigerant`**(*state: ~ctypes.c_void_p*, *api: <ctypes.LibraryLoader object at 0x769a47078950>*, *refrigerant_name: bytes*) [source]

Bases: `object`

This class provides access to the refrigerant property calculations inside EnergyPlus. For now, the only refrigerant name allowed is steam. This is because other refrigerants are only initialized when they are declared in the input file. When calling through the API, there is no input file, so no other refrigerants are declared. This should be improved through later enhancements, but steam will provide a suitable use case for now.

`delete`(*state: c_void_p*) [source]

Frees the memory of the associated Refrigerant instance inside the EnergyPlus (C++) state.

Parameters: **state** – An active EnergyPlus “state” that is returned from a call to *api.state_manager.new_state()*.

Returns: Nothing

`saturated_density`(*state: c_void_p*, *temperature: float*, *quality: float*)→ float [source]

Returns the refrigerant density at the specified temperature and quality.

Parameters:
- **state** – An active EnergyPlus “state” that is returned from a call to *api.state_manager.new_state()*.
- **temperature** – Refrigerant temperature, in Celsius
- **quality** – Refrigerant quality, in fractional form from 0.0 to 1.0

Returns: Refrigerant saturated density, in kg/m3

`saturated_enthalpy`(*state: c_void_p*, *temperature: float*, *quality: float*)→ float [source]

Returns the refrigerant saturated enthalpy at the specified temperature and quality.

Parameters:
- **state** – An active EnergyPlus “state” that is returned from a call to *api.state_manager.new_state()*.
- **temperature** – Refrigerant temperature, in Celsius
- **quality** – Refrigerant quality, in fractional form from 0.0 to 1.0

Returns: Refrigerant saturated enthalpy, in J/kg

`saturated_specific_heat`(*state: c_void_p*, *temperature: float*, *quality: float*)→ float [source]

Returns the refrigerant specific heat at the specified temperature and quality.

Parameters:
- **state** – An active EnergyPlus “state” that is returned from a call to *api.state_manager.new_state()*.
- **temperature** – Refrigerant temperature, in Celsius

- **quality** – Refrigerant quality, in fractional form from 0.0 to 1.0

Returns: Refrigerant saturated specific heat, in J/kg-K

`saturation_pressure`(*state: c_void_p*, *temperature: float*)→ float [source]

Returns the saturation pressure of the refrigerant at the specified temperature.

Parameters:
- **state** – An active EnergyPlus “state” that is returned from a call to *api.state_manager.new_state()*.
- **temperature** – Refrigerant temperature, in Celsius.

Returns: Refrigerant saturation pressure, in Pa

`saturation_temperature`(*state: c_void_p*, *pressure: float*)→ float [source]

Returns the saturation temperature of the refrigerant at the specified pressure.

Parameters:
- **state** – An active EnergyPlus “state” that is returned from a call to *api.state_manager.new_state()*.
- **pressure** – Refrigerant pressure, in Pa

Returns: Refrigerant saturation temperature, in Celsius

Runtime API

class `runtime.Runtime`(*api: <ctypes.LibraryLoader object at 0x769a47078950>*) [source]

Bases: `object`

This API class enables a client to hook into EnergyPlus at runtime and sense/actuate data in a running simulation. The pattern is quite simple: create a callback function in Python, and register it with one of the registration methods on this class to allow the callback to be called at a specific point in the simulation. Inside the callback function, the client can get sensor values and set actuator values using the DataTransfer API methods, and also look up values and perform calculations using EnergyPlus internal methods via the Functional API methods.

`callback_after_component_get_input`(*state: c_void_p*, *f: LambdaType*)→ None [source]

This function allows a client to register a function to be called back by EnergyPlus at the end of component get input processes.

Parameters:
- **state** – An active EnergyPlus “state” that is returned from a call to *api.state_manager.new_state()*.
- **f** – A python function which takes one argument, the current state instance, and returns nothing

Returns: Nothing

`callback_after_new_environment_warmup_complete`(*state: c_void_p*, *f: LambdaType*)→ None [source]

This function allows a client to register a function to be called back by EnergyPlus at the warmup of each environment.

Parameters:
- **state** – An active EnergyPlus “state” that is returned from a call to *api.state_manager.new_state()*.
- **f** – A python function which takes one argument, the current state instance, and returns nothing

Returns: Nothing

`callback_after_predictor_after_hvac_managers`(*state: c_void_p*, *f: LambdaType*)→ None [source]

This function allows a client to register a function to be called back by EnergyPlus at the end of the predictor step after HVAC managers.

Parameters:
- **state** – An active EnergyPlus “state” that is returned from a call to *api.state_manager.new_state()*.
- **f** – A python function which takes one argument, the current state instance, and returns nothing

Returns: Nothing

`callback_after_predictor_before_hvac_managers`(*state: c_void_p*, *f: LambdaType*)→ None [source]

This function allows a client to register a function to be called back by EnergyPlus at the end of the predictor step but before HVAC managers.

Parameters:
- **state** – An active EnergyPlus “state” that is returned from a call to *api.state_manager.new_state()*.
- **f** – A python function which takes one argument, the current state instance, and returns nothing

Returns: Nothing

`callback_begin_new_environment`(*state: c_void_p*, *f: LambdaType*)→ None [source]

This function allows a client to register a function to be called back by EnergyPlus at the beginning of each environment.

Parameters:
- **state** – An active EnergyPlus “state” that is returned from a call to *api.state_manager.new_state()*.
- **f** – A python function which takes one argument, the current state instance, and re

Returns: Nothing

stable

`callback_begin_system_timestep_before_predictor`(*state: c_void_p*, *f: LambdaType*)→ None [source]

This function allows a client to register a function to be called back by EnergyPlus at the beginning of system time step .

Parameters:
- **state** – An active EnergyPlus “state” that is returned from a call to *api.state_manager.new_state()*.
- **f** – A python function which takes one argument, the current state instance, and returns nothing

Returns: Nothing

`callback_begin_zone_timestep_after_init_heat_balance`(*state: c_void_p*, *f: LambdaType*)→ None [source]

This function allows a client to register a function to be called back by EnergyPlus at the beginning of the zone time step after init heat balance.

Parameters:
- **state** – An active EnergyPlus “state” that is returned from a call to *api.state_manager.new_state()*.
- **f** – A python function which takes one argument, the current state instance, and returns nothing

Returns: Nothing

`callback_begin_zone_timestep_before_init_heat_balance`(*state: c_void_p*, *f: LambdaType*)→ None [source]

This function allows a client to register a function to be called back by EnergyPlus at the beginning of the zone time step before init heat balance.

Parameters:
- **state** – An active EnergyPlus “state” that is returned from a call to *api.state_manager.new_state()*.
- **f** – A python function which takes one argument, the current state instance, and returns nothing

Returns: Nothing

`callback_begin_zone_timestep_before_set_current_weather`(*state: c_void_p*, *f: LambdaType*)→ None [source]

This function allows a client to register a function to be called back by EnergyPlus at the beginning of zone time step, before weather is updated.

Parameters:
- **state** – An active EnergyPlus “state” that is returned from a call to *api.state_manager.new_state()*.
- **f** – A python function which takes one argument, the current state instance, and returns nothing

Returns: Nothing

`callback_end_system_sizing`(*state: c_void_p*, *f: LambdaType*)→ None [source]

This function allows a client to register a function to be called back by EnergyPlus at the end of the system sizing process.

Parameters:
- **state** – An active EnergyPlus “state” that is returned from a call to *api.state_manager.new_state()*.
- **f** – A python function which takes one argument, the current state instance, and returns nothing

Returns: Nothing

`callback_end_system_timestep_after_hvac_reporting`(*state: c_void_p*, *f: LambdaType*)→ None [source]

This function allows a client to register a function to be called back by EnergyPlus at the end of a system time step and after HVAC reporting.

Parameters:
- **state** – An active EnergyPlus “state” that is returned from a call to *api.state_manager.new_state()*.
- **f** – A python function which takes one argument, the current state instance, and returns nothing

Returns: Nothing

`callback_end_system_timestep_before_hvac_reporting`(*state: c_void_p*, *f: LambdaType*)→ None [source]

This function allows a client to register a function to be called back by EnergyPlus at the end of a system time step, but before HVAC reporting.

Parameters:
- **state** – An active EnergyPlus “state” that is returned from a call to *api.state_manag* stable
- **f** – A python function which takes one argument, the current state instance, and returns nothing

Returns: Nothing

`callback_end_zone_sizing`(*state: c_void_p, f: LambdaType*)→ None [source]

This function allows a client to register a function to be called back by EnergyPlus at the end of the zone sizing process.

Parameters:
- **state** – An active EnergyPlus "state" that is returned from a call to *api.state_manager.new_state()*.
- **f** – A python function which takes one argument, the current state instance, and returns nothing

Returns: Nothing

`callback_end_zone_timestep_after_zone_reporting`(*state: c_void_p, f: LambdaType*)→ None [source]

This function allows a client to register a function to be called back by EnergyPlus at the end of a zone time step and after zone reporting.

Parameters:
- **state** – An active EnergyPlus "state" that is returned from a call to *api.state_manager.new_state()*.
- **f** – A python function which takes one argument, the current state instance, and returns nothing

Returns: Nothing

`callback_end_zone_timestep_before_zone_reporting`(*state: c_void_p, f: LambdaType*)→ None [source]

This function allows a client to register a function to be called back by EnergyPlus at the end of a zone time step but before zone reporting has been completed.

Parameters:
- **state** – An active EnergyPlus "state" that is returned from a call to *api.state_manager.new_state()*.
- **f** – A python function which takes one argument, the current state instance, and returns nothing

Returns: Nothing

`callback_inside_system_iteration_loop`(*state: c_void_p, f: LambdaType*)→ None [source]

This function allows a client to register a function to be called back by EnergyPlus inside the system iteration loop.

Parameters:
- **state** – An active EnergyPlus "state" that is returned from a call to *api.state_manager.new_state()*.
- **f** – A python function which takes one argument, the current state instance, and returns nothing

Returns: Nothing

`callback_message`(*state: c_void_p, f: LambdaType*)→ None [source]

This function allows a client to register a function to be called back by EnergyPlus when printing anything to standard output. This can allow a GUI to easily show the output of EnergyPlus streaming by. When used in conjunction with the progress callback, a progress bar and status text label can provide a nice EnergyPlus experience on a GUI.

Parameters:
- **state** – An active EnergyPlus "state" that is returned from a call to *api.state_manager.new_state()*.
- **f** – A python function which takes a string (bytes) argument and returns nothing

Returns: Nothing

`callback_progress`(*state: c_void_p, f: LambdaType*)→ None [source]

This function allows a client to register a function to be called back by EnergyPlus at the end of each day with a progress (percentage) indicator

Parameters:
- **state** – An active EnergyPlus "state" that is returned from a call to *api.state_manager.new_state()*.
- **f** – A python function which takes an integer argument and returns nothing

Returns: Nothing

`callback_register_external_hvac_manager`(*state: c_void_p, f: LambdaType*)→ None [source]

This function allows a client to register an external HVAC manager function to be called back in EnergyPlus. By registering this function, EnergyPlus will bypass all HVAC calculations and expect that this function will manage all HVAC through sensors and actuators. Right now this function is not well-supported, and this callback sh stable considered purely as a placeholder until a future release refines the use case.

Parameters:
- **state** – An active EnergyPlus "state" that is returned from a call to *api.state_manager.new_state()*.

- **f** – A python function which takes one argument, the current state instance, and returns nothing

Returns: Nothing

`callback_unitary_system_sizing`(*state: c_void_p*, *f: LambdaType*)→ None [source]

This function allows a client to register a function to be called back by EnergyPlus in unitary system sizing.

Parameters:
- **state** – An active EnergyPlus “state” that is returned from a call to *api.state_manager.new_state()*.
- **f** – A python function which takes one argument, the current state instance, and returns nothing

Returns: Nothing

`callback_user_defined_component_model`(*state: c_void_p*, *f: LambdaType*, *program_name: str*)→ None [source]

This function allows a client to register a function to be called by a specific user-defined equipment object inside EnergyPlus. This registration function takes a string for the program “name” which should match the name given in the IDF.

Parameters:
- **state** – An active EnergyPlus “state” that is returned from a call to *api.state_manager.new_state()*.
- **f** – A python function which takes one argument, the current state instance, and returns nothing.
- **program_name** – The program name which is listed in the IDF on the user-defined object, either as an initialization program name or a simulation program name.

Returns: Nothing

static `clear_callbacks`()→ None [source]

This function is used if you are running this script continually making multiple calls into the E+ library in one thread. EnergyPlus should be cleaned up between runs.

Note this will clean all registered callbacks, so functions must be registered again prior to the next run.

Returns: Nothing

`issue_severe`(*state: c_void_p*, *message: str | bytes*)→ None [source]

This function allows a script to issue an error through normal EnergyPlus methods. The message will be listed in the standard EnergyPlus error file once the simulation is complete. This function has limited usefulness when calling EnergyPlus as a library, as errors can be handled by the calling client, however, in a PythonPlugin workflow, this can be an important tool to alert the user of issues once EnergyPlus(.exe) has finished running.

Note the severe errors tend to lead to EnergyPlus terminating with a Fatal Error. This can be accomplished in PythonPlugin workflows by issuing a severe error, followed by returning 1 from the plugin function. EnergyPlus will interpret this return value as a signal to terminate with a fatal error.

Note that the argument passed in here can be either a string or a bytes object, as this wrapper handles conversion as needed.

Parameters:
- **state** – An active EnergyPlus “state” that is returned from a call to *api.state_manager.new_state()*.
- **message** – The error message to be listed in the error file.

Returns: Nothing

`issue_text`(*state: c_void_p*, *message: str | bytes*)→ None [source]

This function allows a script to issue a message through normal EnergyPlus methods. The message will be listed in the standard EnergyPlus error file once the simulation is complete. This function can be used alongside the warning and error issuance functions to provide further context to the situation. This function has limited usefulness when calling EnergyPlus as a library, as errors can be handled by the calling client, however, in a PythonPlugin workflow, this can be an important tool to alert the user of issues once EnergyPlus(.exe) has finished running.

stable

Note that the argument passed in here can be either a string or a bytes object, as this wrapper handles
needed.

Parameters:
- **state** – An active EnergyPlus “state” that is returned from a call to *api.state_manager.new_state()*.
- **message** – The message to be listed in the error file.

Returns: Nothing

`issue_warning`(*state: c_void_p*, *message: str | bytes*)→ None [source]

This function allows a script to issue a warning through normal EnergyPlus methods. The message will be listed in the standard EnergyPlus error file once the simulation is complete. This function has limited usefulness when calling EnergyPlus as a library, as errors can be handled by the calling client, however, in a PythonPlugin workflow, this can be an important tool to alert the user of issues once EnergyPlus(.exe) has finished running.

Note that the argument passed in here can be either a string or a bytes object, as this wrapper handles conversion as needed.

Parameters:
- **state** – An active EnergyPlus “state” that is returned from a call to *api.state_manager.new_state()*.
- **message** – The warning message to be listed in the error file.

Returns: Nothing

`run_energyplus`(*state: c_void_p*, *command_line_args: List[str | bytes]*)→ int [source]

This function calls EnergyPlus to run a simulation. The C API expects to find arguments matching the command line string when executing EnergyPlus. When calling the C API directly, the client must create a list of char arguments starting with the program name, followed by all the command line options. For this Python API, the program name is not passed in as an argument, rather only the command line options.

An example call: state = api.state_manager.new_state() run_energyplus(state, [‘-d’, ‘/path/to/output/directory’, ‘-w’, ‘/path/to/weather.epw’, ‘/path/to/input.idf’])

Parameters:
- **state** – An active EnergyPlus “state” that is returned from a call to *api.state_manager.new_state()*.
- **command_line_args** – The command line arguments that would be passed into EnergyPlus if executing directly from the EnergyPlus executable.

Returns: An integer exit code from the simulation, zero is success, non-zero is failure

`set_console_output_status`(*state*, *print_output: bool*)→ None [source]

Allows disabling (and enabling) of console output (stdout and stderr) when calling EnergyPlus as a library. :param state: An active EnergyPlus “state” that is returned from a call to *api.state_manager.new_state()*. :param print_output: A boolean flag for whether we should mute console output :return: Nothing

`stop_simulation`(*state: c_void_p*)→ None [source]

Data Transfer API

class `datatransfer.DataExchange`(*api: <ctypes.LibraryLoader object at 0x769a47078950>, running_as_python_plugin: bool = False*) [source]

Bases: `object`

This API class enables data transfer between EnergyPlus and a client. Output variables and meters are treated as "sensor" data. A client should get a handle (integer) using one of the worker methods then make calls to get data using that handle. Some specific variables in EnergyPlus are controllable as actuators. These work the same way, in that a client should get an integer handle to an actuator and then use the workers to set the value on the actuator. There are also some static data members in EnergyPlus that are exposed as "internal" variables. These variables hold static data such as zone floor area and volume, so they do not change during a simulation, but are constant once they are assigned. Even with this difference, the variables are still handled the same way, by getting an integer handle and then accessing the data using that handle.

This data transfer class is used in one of two workflows:

- When an outside tool is already running EnergyPlus using the Runtime API, and data transfer is to be made during callback functions. In this case, the script should create a DataTransfer API class by calling the *data_transfer* method on the main API class, never trying to create this class directly.
- When a Python script is used in the EnergyPlus Python Plugin System, and the user runs the EnergyPlus binary. In this case, the plugin may need access to state data to make control decisions, and this class enables that. The plugin base class automatically creates an instance of this class, so client plugins that inherit that class will have a *self.api.exchange* instance of this class available, and should *not* attempt to create another one.

Client Python code may make use of the methods to get/set plugin global variables, but only in the Python Plugin cases. For the outside tool API usage, plugin global variables are not available, and data should be shared in the outside calling code.

class `APIDataExchangePoint`(*_what: str, _name: str, _key: str, _type: str, _unit: str*) [source]

Bases: `object`

A class of string members that describe a single API exchange point. The exchange described in this class could represent output variables, output meters, actuators, and more. The "type" member variable can be used to filter a specific type.

key: *str*

This represents the unique ID for this exchange point. In the example of the chiller output variable, this could be "Chiller 1". This is not used for meters

name: *str*

This represents the name of the entry point, not the name of the specific instance of the entry point. Some examples of this name could be "Chiller Heat Transfer Rate" – which could be available for multiple chillers.

type: *str*

This represents the "type" of exchange for this exchange point. This is only used for actuators, and represents the control actuation. For a node setpoint actuation, this could be "temperature" or "humidity", for example.

unit: *str*

This represents the unit of measure for this exchange point. This is NOT used for plugin variables suc
PluginGlobalVariable and PluginTrendVariable.

stable

what: *str*

This variable will hold the basic type of API data point, in string form. This can be one of the following: “Actuator”, “InternalVariable”, “PluginGlobalVariable”, “PluginTrendVariable”, “OutputMeter”, or “OutputVariable”. Once the full list of data exchange points are returned from a call to get_api_data, this can be used to quickly filter down to a specific type.

`actual_date_time`(*state: c_void_p*)→ int [source]

Gets a simple sum of the values of the date/time function. Could be used in random seeding.

Parameters: **state** – An active EnergyPlus “state” that is returned from a call to *api.state_manager.new_state()*.
Returns: Integer value of the date/time function.

`actual_time`(*state: c_void_p*)→ int [source]

Gets a simple sum of the values of the time part of the date/time function. Could be used in random seeding.

Parameters: **state** – An active EnergyPlus “state” that is returned from a call to *api.state_manager.new_state()*.
Returns: Integer value of time portion of the date/time function.

`api_data_fully_ready`(*state: c_void_p*)→ bool [source]

Check whether the data exchange API is fully ready. Handles to variables, actuators, and other data are not guaranteed to be defined prior to this being true. Up until this point some output vars, meters, actuators, etc., may not have been registered yet.

Parameters: **state** – An active EnergyPlus “state” that is returned from a call to `api.state_manager.new_state()`.
Returns: Returns a boolean value to indicate whether variables, actuators, and other data are ready for access.

Note

In the case of Surface actuators for “Construction State” and for a call to `get_construction_handle()`, this is not required to be true, as construction data is generally available earlier in the simulation process. Bypassing this check will allow affecting the Sizing calculations.

`api_error_flag`(*state: c_void_p*)→ bool [source]

Check whether the error flag has been activated. A number of functions will return 0 in erroneous situations, and this function allows for disambiguation between valid zero return values and the error condition.

Parameters: **state** – An active EnergyPlus “state” that is returned from a call to *api.state_manager.new_state()*.
Returns: Returns true if the error flag was activated during prior calculations.

`calendar_year`(*state: c_void_p*)→ int [source]

Get the “current” calendar year of the simulation.

Only valid for weather file run periods.

All simulations operate at a real year, either user specified or automatically selected by EnergyPlus based on other data (start day of week + leap year option).

Parameters: **state** – An active EnergyPlus “state” that is returned from a call to *api.state_manager.new_state()*.
Returns: An integer year (2020, for example)

`current_environment_num`(*state: c_void_p*)→ int [source]

Gets the current environment index. EnergyPlus environments are design days, run periods, etc. This function is only expected to be useful in very specialized applications where you control the environment order carefully

stable

Parameters: **state** – An active EnergyPlus “state” that is returned from a call to *api.state_manager.new_state()*.
Returns: The current environment number.

`current_sim_time`(*state: c_void_p*)→ float [source]

Returns the cumulative simulation time from the start of the environment, in hours

Parameters:	**state** – An active EnergyPlus “state” that is returned from a call to *api.state_manager.new_state()*.
Returns:	Value of the simulation time from the start of the environment in fractional hours

`current_time`(*state: c_void_p*)→ float [source]

Get the current time of day in hours, where current time represents the end time of the current time step.

Parameters:	**state** – An active EnergyPlus “state” that is returned from a call to *api.state_manager.new_state()*.
Returns:	A floating point representation of the current time in hours

`day_of_month`(*state: c_void_p*)→ int [source]

Get the current day of month (1-31)

Parameters:	**state** – An active EnergyPlus “state” that is returned from a call to *api.state_manager.new_state()*.
Returns:	An integer day of the month (1-31)

`day_of_week`(*state: c_void_p*)→ int [source]

Get the current day of the week (1-7)

Parameters:	**state** – An active EnergyPlus “state” that is returned from a call to *api.state_manager.new_state()*.
Returns:	An integer day of week (1-7)

`day_of_year`(*state: c_void_p*)→ int [source]

Get the current day of the year (1-366)

Parameters:	**state** – An active EnergyPlus “state” that is returned from a call to *api.state_manager.new_state()*.
Returns:	An integer day of the year (1-366)

`daylight_savings_time_indicator`(*state: c_void_p*)→ bool [source]

Get the current daylight savings time indicator as a logical value. The C API returns an integer where 1 is yes and 0 is no, this simply wraps that with a bool conversion.

Parameters:	**state** – An active EnergyPlus “state” that is returned from a call to *api.state_manager.new_state()*.
Returns:	A boolean DST indicator for the current time.

`get_actuator_handle`(*state: c_void_p, component_type: str | bytes, control_type: str | bytes, actuator_key: str | bytes*)→ int [source]

Get a handle to an available actuator in a running simulation.

The arguments passed into this function do not need to be a particular case, as the EnergyPlus API automatically converts values to upper-case when finding matches to internal variables in the simulation.

Note also that the arguments passed in here can be either strings or bytes, as this wrapper handles conversion as needed.

Parameters:

- **state** – An active EnergyPlus “state” that is returned from a call to *api.state_manager.new_state()*.
- **component_type** – The actuator category, e.g. “Weather Data”
- **control_type** – The name of the actuator to retrieve, e.g. “Outdoor Dew Point”
- **actuator_key** – The instance of the variable to retrieve, e.g. “Environment”

Returns: An integer ID for this output variable, or -1 if one could not be found.

`get_actuator_value`(*state: c_void_p, actuator_handle: int*)→ float [source]

stable

Gets the most recent value of an actuator. In some applications, actuators are altered by multiple scripts, and this allows getting the most recent value.

Parameters:
- **state** – An active EnergyPlus “state” that is returned from a call to *api.state_manager.new_state()*.
- **actuator_handle** – An integer returned from the *get_actuator_handle* function.

Returns: A floating point of the actuator value. For boolean actuators returns 1.0 for true and 0.0 for false. Returns zero if the handle is invalid. Use the api_error_flag function to disambiguate between valid zero returns and error states.

`get_api_data`(*state: c_void_p*)→ List[APIDataExchangePoint] [source]

Returns a nicely formed list of API data exchange points available in the current simulation.

Parameters: **state** – An active EnergyPlus “state” that is returned from a call to *api.state_manager.new_state()*.

Returns: Returns a Python list of APIDataExchangePoint instances, which can be filtered to inspect exchanges.

`get_construction_handle`(*state: c_void_p, var_name: str | bytes*)→ int [source]

Get a handle to a constructions in a running simulation.

Some actuators allow specifying different constructions to allow switchable construction control. This function returns an index that can be used in those functions. The construction is specified by name.

The arguments passed into this function do not need to be a particular case, as the EnergyPlus API automatically converts values to upper-case when finding matches to internal variables in the simulation.

Note also that the arguments passed in here can be either strings or bytes, as this wrapper handles conversion as needed.

Parameters:
- **state** – An active EnergyPlus “state” that is returned from a call to *api.state_manager.new_state()*.
- **var_name** – Name of the construction to look up

Returns: An integer ID for this construction, or -1 if one could not be found.

`get_ems_global_handle`(*state: c_void_p, var_name: str | bytes*)→ int [source]

Get a handle to an EMS global variable in a running simulation.

EMS global variables are used as a way to share data between running EMS programs. First a global variable must be declared in the input file using the EnergyManagementSystem:GlobalVariable object. Once a name has been declared, it can be accessed by EMS programs by name, and through the Python API. For API usage, the client should get a handle to the variable using this get_global_handle function, then using the get_ems_global_value and set_ems_global_value functions as needed. Note all global variables are floating point values.

The arguments passed into this function do not need to be a particular case, as the EnergyPlus API automatically converts values to upper-case when finding matches to internal variables in the simulation.

Note also that the arguments passed in here can be either strings or bytes, as this wrapper handles conversion as needed.

Parameters:
- **state** – An active EnergyPlus “state” that is returned from a call to *api.state_manager.new_state()*.
- **var_name** – The name of the EMS global variable to retrieve, this name must be listed in an IDF object: *EnergyManagementSystem:GlobalVariable*

Returns: An integer ID for this EMS global variable, or -1 if one could not be found.

`get_ems_global_value`(*state: c_void_p, handle: int*)→ float [source]

Get the current value of an EMS global variable in a running simulation.

EMS global variables are used as a way to share data between running EMS programs. First a global vari
declared in the input file using the EnergyManagementSystem:GlobalVariable object. Once a name has l
can be accessed by EMS programs by name, and through the Python API. For API usage, the client should get a handle to

stable

the variable using this get_global_handle function, then using the get_ems_global_value and set_ems_global_value functions as needed. Note all global variables are floating point values.

Parameters:
- **state** – An active EnergyPlus "state" that is returned from a call to *api.state_manager.new_state()*.
- **handle** – An integer returned from the *get_ems_global_handle* function.

Returns: Floating point representation of the EMS global variable value

`get_global_handle`(*state: c_void_p*, *var_name: str | bytes*)→ int [source]

Get a handle to a global variable in a running simulation. This is only used for Python Plugin applications!

Global variables are used as a way to share data between running Python Plugins. First a global variable must be declared in the input file using the PythonPlugin:GlobalVariables object. Once a name has been declared, it can be accessed in the Plugin by getting a handle to the variable using this get_global_handle function, then using the get_global_value and set_global_value functions as needed. Note all global variables are floating point values.

The arguments passed into this function do not need to be a particular case, as the EnergyPlus API automatically converts values to upper-case when finding matches to internal variables in the simulation.

Note also that the arguments passed in here can be either strings or bytes, as this wrapper handles conversion as needed.

Parameters:
- **state** – An active EnergyPlus "state" that is returned from a call to *api.state_manager.new_state()*.
- **var_name** – The name of the global variable to retrieve, this name must be listed in the IDF object: *PythonPlugin:GlobalVariables*

Returns: An integer ID for this global variable, or -1 if one could not be found.

`get_global_value`(*state: c_void_p*, *handle: int*)→ float [source]

Get the current value of a plugin global variable in a running simulation. This is only used for Python Plugin applications!

Global variables are used as a way to share data between running Python Plugins. First a global variable must be declared in the input file using the PythonPlugin:GlobalVariables object. Once a name has been declared, it can be accessed in the Plugin by getting a handle to the variable using the get_global_handle function, then using this get_global_value and the set_global_value functions as needed. Note all global variables are floating point values.

Parameters:
- **state** – An active EnergyPlus "state" that is returned from a call to *api.state_manager.new_state()*.
- **handle** – An integer returned from the *get_global_handle* function.

Returns: Floating point representation of the global variable value

`get_input_file_path`(*state: c_void_p*)→ Path [source]

Provides the input file path back to the client. In most circumstances the client will know the path to the input file, but there are some cases where code is generalized in unexpected workflows. Users have requested a way to get the input file path back from the running instance.

Parameters: **state** – An active EnergyPlus "state" that is returned from a call to *api.state_manager.new_state()*.

Returns: A pathlib.Path of the input file path

`get_internal_variable_handle`(*state: c_void_p*, *variable_type: str | bytes*, *variable_key: str | bytes*)→ int [source]

Get a handle to an internal variable in a running simulation.

The arguments passed into this function do not need to be a particular case, as the EnergyPlus API automatically converts values to upper-case when finding matches to internal variables in the simulation.

Note also that the arguments passed in here can be either strings or bytes, as this wrapper handles conversion as needed.

stable

Parameters:
- **state** – An active EnergyPlus "state" that is returned from a call to *api.state_manag*
- **variable_type** – The name of the variable to retrieve, e.g. "Zone Air Volume", or "Zone Floor Area"

- **variable_key** – The instance of the variable to retrieve, e.g. “Zone 1”

Returns: An integer ID for this output variable, or -1 if one could not be found.

`get_internal_variable_value`(*state: c_void_p*, *variable_handle: int*)→ float [source]

Get the value of an internal variable in a running simulation. The *get_internal_variable_handle* function is first used to get a handle to the variable by name. Then once the handle is retrieved, it is passed into this function to then get the value of the variable.

Parameters:
- **state** – An active EnergyPlus “state” that is returned from a call to *api.state_manager.new_state()*.
- **variable_handle** – An integer returned from the *get_internal_variable_handle* function.

Returns: Floating point representation of the internal variable value. Returns zero if the handle is invalid. Use the api_error_flag function to disambiguate between valid zero returns and error states.

`get_meter_handle`(*state: c_void_p*, *meter_name: str | bytes*)→ int [source]

Get a handle to a meter in a running simulation.

The meter name passed into this function do not need to be a particular case, as the EnergyPlus API automatically converts values to upper-case when finding matches to internal variables in the simulation.

Note also that the meter name passed in here can be either strings or bytes, as this wrapper handles conversion as needed.

Parameters:
- **state** – An active EnergyPlus “state” that is returned from a call to *api.state_manager.new_state()*.
- **meter_name** – The name of the variable to retrieve, e.g. “Electricity:Facility”, or “Fans:Electricity”

Returns: An integer ID for this meter, or -1 if one could not be found.

`get_meter_value`(*state: c_void_p*, *meter_handle: int*)→ float [source]

Get the current value of a meter in a running simulation. The *get_meter_handle* function is first used to get a handle to the meter by name. Then once the handle is retrieved, it is passed into this function to then get the value of the meter.

Caution: This function currently returns the instantaneous value of a meter, not the cumulative value. This will change in a future version of the API.

Parameters:
- **state** – An active EnergyPlus “state” that is returned from a call to *api.state_manager.new_state()*.
- **meter_handle** – An integer returned from the *get_meter_handle* function.

Returns: Floating point representation of the current meter value. Returns zero if the handle is invalid. Use the api_error_flag function to disambiguate between valid zero returns and error states.

`get_num_nodes_in_cond_fd_surf_layer`(*state: c_void_p*, *surface_name: str | bytes*, *material_name: str | bytes*)→ None [source]

Get the number of nodes in CondFD surface layer.

Parameters:
- **state** – An active EnergyPlus “state” that is returned from a call to *api.state_manager.new_state()*.
- **surface_name** – The name of the surface as defined in the input file, such as “ZN001:Surf001”.
- **material_name** – The name of the surface material layer as defined in the input file, such as “GypsumBoardLayer”.

Returns: Nothing

`get_object_names`(*state: c_void_p*, *object_type_name: str | bytes*)→ List[str] [source]

Gets the instance names for a given object type in the current input file :param state: An active EnergyPlus “state” that is returned from a call to *api.state_manager.new_state()*. :param object_type_name: The object type name as defined in the schema, such as “Zone” or “Chiller:Electric” :return: A list of strings represented the names of the objects in the input file

`get_trend_average`(*state: c_void_p*, *trend_handle: int*, *count: int*)→ float [source]

stable

Get the average of a plugin trend variable over a specific history set. The count argument specifies how many time steps to go back in the trend history. A value of 1 indicates averaging just the most recent value. The value of time_index must be less than or equal to the number of history terms specified in the matching PythonPlugin:TrendVariable object declaration in the input file. This is only used for Python Plugin applications!

Trend variables are used as a way to track history of a PythonPlugin:Variable over time. First a trend variable must be declared in the input file using the PythonPlugin:TrendVariable object. Once a variable has been declared there, it can be accessed in the Plugin by getting a handle to the variable using the get_trend_handle function, then using the other trend variable worker functions as needed.

Parameters:
- **state** – An active EnergyPlus "state" that is returned from a call to *api.state_manager.new_state()*.
- **trend_handle** – An integer returned from the *get_trend_handle* function.
- **count** – The number of time steps to search back in history to evaluate this function.

Returns: Floating point value representation of the specific evaluation.

`get_trend_direction`(*state: c_void_p*, *trend_handle: int*, *count: int*)→ float [source]

Get the trajectory of a plugin trend variable over a specific history set. The count argument specifies how many time steps to go back in the trend history. A value of 1 indicates sweeping just the most recent value. A linear regression is performed over the swept values and the slope of the regression line is returned as a representation of the average trajectory over this range. The value of time_index must be less than or equal to the number of history terms specified in the matching PythonPlugin:TrendVariable object declaration in the input file. This is only used for Python Plugin applications!

Trend variables are used as a way to track history of a PythonPlugin:Variable over time. First a trend variable must be declared in the input file using the PythonPlugin:TrendVariable object. Once a variable has been declared there, it can be accessed in the Plugin by getting a handle to the variable using the get_trend_handle function, then using the other trend variable worker functions as needed.

Parameters:
- **state** – An active EnergyPlus "state" that is returned from a call to *api.state_manager.new_state()*.
- **trend_handle** – An integer returned from the *get_trend_handle* function.
- **count** – The number of time steps to search back in history to evaluate this function.

Returns: Floating point value representation of the specific evaluation.

`get_trend_handle`(*state: c_void_p*, *trend_var_name: str | bytes*)→ int [source]

Get a handle to a trend variable in a running simulation. This is only used for Python Plugin applications!

Trend variables are used as a way to track history of a PythonPlugin:Variable over time. First a trend variable must be declared in the input file using the PythonPlugin:TrendVariable object. Once a variable has been declared there, it can be accessed in the Plugin by getting a handle to the variable using this get_trend_handle function, then using the other trend variable worker functions as needed.

The arguments passed into this function do not need to be a particular case, as the EnergyPlus API automatically converts values to upper-case when finding matches to internal variables in the simulation.

Note also that the arguments passed in here can be either strings or bytes, as this wrapper handles conversion as needed.

Parameters:
- **state** – An active EnergyPlus "state" that is returned from a call to *api.state_manager.new_state()*.
- **trend_var_name** – The name of the global variable to retrieve, this name must match the name of a *PythonPlugin:TrendVariable* IDF object.

Returns: An integer ID for this trend variable, or -1 if one could not be found.

`get_trend_max`(*state: c_void_p*, *trend_handle: int*, *count: int*)→ float [source]

Get the maximum of a plugin trend variable over a specific history set. The count argument specifies ho
steps to go back in the trend history. A value of 1 indicates sweeping just the most recent value. The val
must be less than or equal to the number of history terms specified in the matching PythonPlugin:TrendVariable object

stable

declaration in the input file. This is only used for Python Plugin applications!

Trend variables are used as a way to track history of a PythonPlugin:Variable over time. First a trend variable must be declared in the input file using the PythonPlugin:TrendVariable object. Once a variable has been declared there, it can be accessed in the Plugin by getting a handle to the variable using the get_trend_handle function, then using the other trend variable worker functions as needed.

Parameters:
- **state** – An active EnergyPlus "state" that is returned from a call to *api.state_manager.new_state()*.
- **trend_handle** – An integer returned from the *get_trend_handle* function.
- **count** – The number of time steps to search back in history to evaluate this function.

Returns: Floating point value representation of the specific evaluation.

`get_trend_min`(*state: c_void_p*, *trend_handle: int*, *count: int*)→ float [source]

Get the minimum of a plugin trend variable over a specific history set. The count argument specifies how many time steps to go back in the trend history. A value of 1 indicates sweeping just the most recent value. The value of time_index must be less than or equal to the number of history terms specified in the matching PythonPlugin:TrendVariable object declaration in the input file. This is only used for Python Plugin applications!

Trend variables are used as a way to track history of a PythonPlugin:Variable over time. First a trend variable must be declared in the input file using the PythonPlugin:TrendVariable object. Once a variable has been declared there, it can be accessed in the Plugin by getting a handle to the variable using the get_trend_handle function, then using the other trend variable worker functions as needed.

Parameters:
- **state** – An active EnergyPlus "state" that is returned from a call to *api.state_manager.new_state()*.
- **trend_handle** – An integer returned from the *get_trend_handle* function.
- **count** – The number of time steps to search back in history to evaluate this function.

Returns: Floating point value representation of the specific evaluation.

`get_trend_sum`(*state: c_void_p*, *trend_handle: int*, *count: int*)→ float [source]

Get the summation of a plugin trend variable over a specific history set. The count argument specifies how many time steps to go back in the trend history. A value of 1 indicates sweeping just the most recent value. The value of time_index must be less than or equal to the number of history terms specified in the matching PythonPlugin:TrendVariable object declaration in the input file. This is only used for Python Plugin applications!

Trend variables are used as a way to track history of a PythonPlugin:Variable over time. First a trend variable must be declared in the input file using the PythonPlugin:TrendVariable object. Once a variable has been declared there, it can be accessed in the Plugin by getting a handle to the variable using the get_trend_handle function, then using the other trend variable worker functions as needed.

Parameters:
- **state** – An active EnergyPlus "state" that is returned from a call to *api.state_manager.new_state()*.
- **trend_handle** – An integer returned from the *get_trend_handle* function.
- **count** – The number of time steps to search back in history to evaluate this function.

Returns: Floating point value representation of the specific evaluation.

`get_trend_value`(*state: c_void_p*, *trend_handle: int*, *time_index: int*)→ float [source]

Get the value of a plugin trend variable at a specific history point. The time_index argument specifies how many time steps to go back in the trend history. A value of 1 indicates taking the most recent value. The value of time_index must be less than or equal to the number of history terms specified in the matching PythonPlugin:TrendVariable object declaration in the input file. This is only used for Python Plugin applications!

Trend variables are used as a way to track history of a PythonPlugin:Variable over time. First a trend variable must be declared in the input file using the PythonPlugin:TrendVariable object. Once a variable has been declared there, it can be accessed in the Plugin by getting a handle to the variable using the get_trend_handle function, then using the other trend variable worker functions as needed.

stable

Parameters:
- **state** – An active EnergyPlus "state" that is returned from a call to *api.state_manager.new_state()*.

- **trend_handle** – An integer returned from the *get_trend_handle* function.
- **time_index** – The number of time steps to search back in history to evaluate this function.

Returns: Floating point value representation of the specific evaluation.

`get_variable_handle`(*state: c_void_p*, *variable_name: str | bytes*, *variable_key: str | bytes*)→ int [source]

Get a handle to an output variable in a running simulation.

The arguments passed into this function do not need to be a particular case, as the EnergyPlus API automatically converts values to upper-case when finding matches to internal variables in the simulation.

Note also that the arguments passed in here can be either strings or bytes, as this wrapper handles conversion as needed.

Parameters:
- **state** – An active EnergyPlus “state” that is returned from a call to *api.state_manager.new_state()*.
- **variable_name** – The name of the variable to retrieve, e.g. “Site Outdoor Air DryBulb Temperature”, or “Fan Air Mass Flow Rate”
- **variable_key** – The instance of the variable to retrieve, e.g. “Environment”, or “Main System Fan”

Returns: An integer ID for this output variable, or -1 if one could not be found.

`get_variable_value`(*state: c_void_p*, *variable_handle: int*)→ float [source]

Get the current value of a variable in a running simulation. The *get_variable_handle* function is first used to get a handle to the variable by name. Then once the handle is retrieved, it is passed into this function to then get the value of the variable.

Parameters:
- **state** – An active EnergyPlus “state” that is returned from a call to *api.state_manager.new_state()*.
- **variable_handle** – An integer returned from the *get_variable_handle* function.

Returns: Floating point representation of the current variable value. Returns zero if the handle is invalid. Use the api_error_flag function to disambiguate between valid zero returns and error states.

`get_weather_file_path`(*state: c_void_p*)→ Path [source]

Provides the weather file path back to the client. In most circumstances the client will know the path to the weather file, but there are some cases where code is generalized in unexpected workflows. Users have requested a way to get the weather file path back from the running instance.

Parameters: **state** – An active EnergyPlus “state” that is returned from a call to *api.state_manager.new_state()*.

Returns: A pathlib.Path of the weather file

`holiday_index`(*state: c_void_p*)→ int [source]

Gets a flag for the current day holiday type: 0 is no holiday, 1 is holiday type #1, etc.

Parameters: **state** – An active EnergyPlus “state” that is returned from a call to *api.state_manager.new_state()*.

Returns: An integer indicator for current day holiday type.

`hour`(*state: c_void_p*)→ int [source]

Get the current hour of the simulation (0-23)

Parameters: **state** – An active EnergyPlus “state” that is returned from a call to *api.state_manager.new_state()*.

Returns: An integer hour of the day (0-23)

`is_raining`(*state: c_void_p*)→ bool [source]

Gets a flag for whether the it is currently raining. The C API returns an integer where 1 is yes and 0 is no, this simply wraps that with a bool conversion.

stable

Parameters: **state** – An active EnergyPlus “state” that is returned from a call to *api.state_manager.new_state()*.

Returns: A boolean indicating whether it is currently raining.

kind_of_sim(*state: c_void_p*)→ int [source]

Gets the current environment number.

Parameters: **state** – An active EnergyPlus “state” that is returned from a call to *api.state_manager.new_state()*.
Returns: Integer value of current environment.

list_available_api_data_csv(*state: c_void_p*)→ bytes [source]

Lists out all API data stuff in an easily parseable CSV form

Parameters: **state** – An active EnergyPlus “state” that is returned from a call to *api.state_manager.new_state()*.
Returns: Returns a raw bytes CSV representation of the available API data

minutes(*state: c_void_p*)→ int [source]

Get the current minutes into the hour (1-60)

Parameters: **state** – An active EnergyPlus “state” that is returned from a call to *api.state_manager.new_state()*.
Returns: An integer number of minutes into the current hour (1-60)

month(*state: c_void_p*)→ int [source]

Get the current month of the simulation (1-12)

Parameters: **state** – An active EnergyPlus “state” that is returned from a call to *api.state_manager.new_state()*.
Returns: An integer month (1-12)

num_time_steps_in_hour(*state: c_void_p*)→ int [source]

Returns the number of zone time steps in an hour, which is currently a constant value throughout a simulation.

Parameters: **state** – An active EnergyPlus “state” that is returned from a call to *api.state_manager.new_state()*.
Returns: An integer representation of the number of time steps in an hour

request_variable(*state: c_void_p, variable_name: str | bytes, variable_key: str | bytes*)→ None [source]

Request output variables so they can be accessed during a simulation.

In EnergyPlus, not all variables are available by default. If they were all available, there would be a terrible memory impact. Instead, only requested and necessary variables are kept in memory. When running EnergyPlus as a program, including when using Python Plugins, variables are requested through input objects. When running EnergyPlus as a library, variables can also be requested through this function call. This function has the same signature as the get_variable_handle function, which is used to then request the ID of a variable once the simulation has begun. NOTE: Variables should be requested before *each* run of EnergyPlus, as the internal array is cleared when clearing the state of each run.

Parameters:

- **state** – An active EnergyPlus “state” that is returned from a call to *api.state_manager.new_state()*.
- **variable_name** – The name of the variable to retrieve, e.g. “Site Outdoor Air DryBulb Temperature”, or “Fan Air Mass Flow Rate”
- **variable_key** – The instance of the variable to retrieve, e.g. “Environment”, or “Main System Fan”

Returns: Nothing

reset_actuator(*state: c_void_p, actuator_handle: int*)→ None [source]

Resets the actuator internally to EnergyPlus. This allows subsequent calculations to be used for the actuator instead of the externally set actuator value.

Parameters:

- **state** – An active EnergyPlus “state” that is returned from a call to *api.state_manag*
- **actuator_handle** – An integer returned from the *get_actuator_handle* function.

Returns: Nothing

stable

`reset_api_error_flag`(*state: c_void_p*)→ None [source]

Resets the error flag for API calls. A number of functions will return 0 in erroneous situations, but activate an error flag. In certain work flows, it may be useful to reset this error flag (unit testing, etc.). This function allows resetting it to false.

Parameters: **state** – An active EnergyPlus “state” that is returned from a call to *api.state_manager.new_state()*.

`set_actuator_value`(*state: c_void_p, actuator_handle: int, actuator_value: float*)→ None [source]

Sets the value of an actuator in a running simulation. The *get_actuator_handle* function is first used to get a handle to the actuator by name. Then once the handle is retrieved, it is passed into this function, along with the value to assign, to then set the value of the actuator. Internally, actuators can alter floating point, integer, and boolean operational values. The API only exposes this set function with a floating point argument. For floating point types, the value is assigned directly. For integer types, the value is rounded to the nearest integer, with the halfway point rounded away from zero (2.5 becomes 3), then cast to a plain integer. For logical values, the original EMS convention is kept, where a value of 1.0 means TRUE, and a value of 0.0 means FALSE – and any other value defaults to FALSE. A small tolerance is applied internally to allow for small floating point round-off. A value *very close* to 1.0 will still evaluate to TRUE.

Parameters:
- **state** – An active EnergyPlus “state” that is returned from a call to *api.state_manager.new_state()*.
- **actuator_handle** – An integer returned from the *get_actuator_handle* function.
- **actuator_value** – The floating point value to assign to the actuator

Returns: Nothing

`set_ems_global_value`(*state: c_void_p, handle: int, value: float*)→ None [source]

Set the current value of an EMS global variable in a running simulation.

EMS global variables are used as a way to share data between running EMS programs. First a global variable must be declared in the input file using the EnergyManagementSystem:GlobalVariable object. Once a name has been declared, it can be accessed by EMS programs by name, and through the Python API. For API usage, the client should get a handle to the variable using this get_global_handle function, then using the get_ems_global_value and set_ems_global_value functions as needed. Note all global variables are floating point values.

Parameters:
- **state** – An active EnergyPlus “state” that is returned from a call to *api.state_manager.new_state()*.
- **handle** – An integer returned from the *get_ems_global_handle* function.
- **value** – Floating point value to assign to the EMS global variable

`set_global_value`(*state: c_void_p, handle: int, value: float*)→ None [source]

Set the current value of a plugin global variable in a running simulation. This is only used for Python Plugin applications!

Global variables are used as a way to share data between running Python Plugins. First a global variable must be declared in the input file using the PythonPlugin:GlobalVariables object. Once a name has been declared, it can be accessed in the Plugin by getting a handle to the variable using the get_global_handle function, then using the get_global_value and this set_global_value functions as needed. Note all global variables are floating point values.

Parameters:
- **state** – An active EnergyPlus “state” that is returned from a call to *api.state_manager.new_state()*.
- **handle** – An integer returned from the *get_global_handle* function.
- **value** – Floating point value to assign to the global variable

`sun_is_up`(*state: c_void_p*)→ bool [source]

Gets a flag for whether the sun is currently up. The C API returns an integer where 1 is yes and 0 is no, this simply wraps that with a bool conversion.

Parameters: **state** – An active EnergyPlus “state” that is returned from a call to *api.state_manager.new_state()*.
Returns: A boolean indicating whether the sun is currently up.

stable

`system_time_step`(*state: c_void_p*)→ float [source]

Gets the current system time step value in EnergyPlus. The system time step is variable and fluctuates during the simulation.

Parameters: **state** – An active EnergyPlus “state” that is returned from a call to *api.state_manager.new_state()*.
Returns: The current system time step in fractional hours.

`today_weather_albedo_at_time`(*state: c_void_p*, *hour: int*, *time_step_number: int*)→ float [source]

Gets the specified weather data at the specified hour and time step index within that hour

Parameters:
- **state** – An active EnergyPlus “state” that is returned from a call to *api.state_manager.new_state()*.
- **hour** – Integer hour of day (0 to 23)
- **time_step_number** – Time step index in hour, from 1 to the number of zone time steps per hour

Returns: Value of the weather condition at the specified time

`today_weather_beam_solar_at_time`(*state: c_void_p*, *hour: int*, *time_step_number: int*)→ float [source]

Gets the specified weather data at the specified hour and time step index within that hour

Parameters:
- **state** – An active EnergyPlus “state” that is returned from a call to *api.state_manager.new_state()*.
- **hour** – Integer hour of day (0 to 23)
- **time_step_number** – Time step index in hour, from 1 to the number of zone time steps per hour

Returns: Value of the weather condition at the specified time

`today_weather_diffuse_solar_at_time`(*state: c_void_p*, *hour: int*, *time_step_number: int*)→ float [source]

Gets the specified weather data at the specified hour and time step index within that hour

Parameters:
- **state** – An active EnergyPlus “state” that is returned from a call to *api.state_manager.new_state()*.
- **hour** – Integer hour of day (0 to 23)
- **time_step_number** – Time step index in hour, from 1 to the number of zone time steps per hour

Returns: Value of the weather condition at the specified time

`today_weather_horizontal_ir_at_time`(*state: c_void_p*, *hour: int*, *time_step_number: int*)→ float [source]

Gets the specified weather data at the specified hour and time step index within that hour

Parameters:
- **state** – An active EnergyPlus “state” that is returned from a call to *api.state_manager.new_state()*.
- **hour** – Integer hour of day (0 to 23)
- **time_step_number** – Time step index in hour, from 1 to the number of zone time steps per hour

Returns: Value of the weather condition at the specified time

`today_weather_is_raining_at_time`(*state: c_void_p*, *hour: int*, *time_step_number: int*)→ bool [source]

Gets the specified weather data at the specified hour and time step index within that hour

Parameters:
- **state** – An active EnergyPlus “state” that is returned from a call to *api.state_manager.new_state()*.
- **hour** – Integer hour of day (0 to 23)
- **time_step_number** – Time step index in hour, from 1 to the number of zone time steps per hour

Returns: A true/false for whether the weather condition is active at the specified time

`today_weather_is_snowing_at_time`(*state: c_void_p*, *hour: int*, *time_step_number: int*)→ bool [source]

Gets the specified weather data at the specified hour and time step index within that hour

Parameters:
- **state** – An active EnergyPlus “state” that is returned from a call to *api.state_manager.new_state()*.
- **hour** – Integer hour of day (0 to 23)
- **time_step_number** – Time step index in hour, from 1 to the number of zone time s

Returns: A true/false for whether the weather condition is active at the specified time

stable

`today_weather_liquid_precipitation_at_time`(*state: c_void_p*, *hour: int*, *time_step_number: int*)→ float [source]

Gets the specified weather data at the specified hour and time step index within that hour

Parameters:
- **state** – An active EnergyPlus "state" that is returned from a call to *api.state_manager.new_state()*.
- **hour** – Integer hour of day (0 to 23)
- **time_step_number** – Time step index in hour, from 1 to the number of zone time steps per hour

Returns: Value of the weather condition at the specified time

`today_weather_outdoor_barometric_pressure_at_time`(*state: c_void_p*, *hour: int*, *time_step_number: int*)→ float [source]

Gets the specified weather data at the specified hour and time step index within that hour

Parameters:
- **state** – An active EnergyPlus "state" that is returned from a call to *api.state_manager.new_state()*.
- **hour** – Integer hour of day (0 to 23)
- **time_step_number** – Time step index in hour, from 1 to the number of zone time steps per hour

Returns: Value of the weather condition at the specified time

`today_weather_outdoor_dew_point_at_time`(*state: c_void_p*, *hour: int*, *time_step_number: int*)→ float [source]

Gets the specified weather data at the specified hour and time step index within that hour

Parameters:
- **state** – An active EnergyPlus "state" that is returned from a call to *api.state_manager.new_state()*.
- **hour** – Integer hour of day (0 to 23)
- **time_step_number** – Time step index in hour, from 1 to the number of zone time steps per hour

Returns: Value of the weather condition at the specified time

`today_weather_outdoor_dry_bulb_at_time`(*state: c_void_p*, *hour: int*, *time_step_number: int*)→ float [source]

Gets the specified weather data at the specified hour and time step index within that hour

Parameters:
- **state** – An active EnergyPlus "state" that is returned from a call to *api.state_manager.new_state()*.
- **hour** – Integer hour of day (0 to 23)
- **time_step_number** – Time step index in hour, from 1 to the number of zone time steps per hour

Returns: Value of the weather condition at the specified time

`today_weather_outdoor_relative_humidity_at_time`(*state: c_void_p*, *hour: int*, *time_step_number: int*)→ float [source]

Gets the specified weather data at the specified hour and time step index within that hour

Parameters:
- **state** – An active EnergyPlus "state" that is returned from a call to *api.state_manager.new_state()*.
- **hour** – Integer hour of day (0 to 23)
- **time_step_number** – Time step index in hour, from 1 to the number of zone time steps per hour

Returns: Value of the weather condition at the specified time

`today_weather_sky_temperature_at_time`(*state: c_void_p*, *hour: int*, *time_step_number: int*)→ float [source]

Gets the specified weather data at the specified hour and time step index within that hour

Parameters:
- **state** – An active EnergyPlus "state" that is returned from a call to *api.state_manager.new_state()*.
- **hour** – Integer hour of day (0 to 23)
- **time_step_number** – Time step index in hour, from 1 to the number of zone time steps per hour

Returns: Value of the weather condition at the specified time

`today_weather_wind_direction_at_time`(*state: c_void_p*, *hour: int*, *time_step_number: int*)→ float [source]

Gets the specified weather data at the specified hour and time step index within that hour

Parameters:
- **state** – An active EnergyPlus "state" that is returned from a call to *api.state_manag*
- **hour** – Integer hour of day (0 to 23)
- **time_step_number** – Time step index in hour, from 1 to the number of zone time steps per hour

stable

Returns: Value of the weather condition at the specified time

`today_weather_wind_speed_at_time`(*state: c_void_p*, *hour: int*, *time_step_number: int*)→ float [source]

Gets the specified weather data at the specified hour and time step index within that hour

Parameters:
- **state** – An active EnergyPlus "state" that is returned from a call to *api.state_manager.new_state()*.
- **hour** – Integer hour of day (0 to 23)
- **time_step_number** – Time step index in hour, from 1 to the number of zone time steps per hour

Returns: Value of the weather condition at the specified time

`tomorrow_weather_albedo_at_time`(*state: c_void_p*, *hour: int*, *time_step_number: int*)→ float [source]

Gets the specified weather data at the specified hour and time step index within that hour

Parameters:
- **state** – An active EnergyPlus "state" that is returned from a call to *api.state_manager.new_state()*.
- **hour** – Integer hour of day (0 to 23)
- **time_step_number** – Time step index in hour, from 1 to the number of zone time steps per hour

Returns: Value of the weather condition at the specified time

`tomorrow_weather_beam_solar_at_time`(*state: c_void_p*, *hour: int*, *time_step_number: int*)→ float [source]

Gets the specified weather data at the specified hour and time step index within that hour

Parameters:
- **state** – An active EnergyPlus "state" that is returned from a call to *api.state_manager.new_state()*.
- **hour** – Integer hour of day (0 to 23)
- **time_step_number** – Time step index in hour, from 1 to the number of zone time steps per hour

Returns: Value of the weather condition at the specified time

`tomorrow_weather_diffuse_solar_at_time`(*state: c_void_p*, *hour: int*, *time_step_number: int*)→ float [source]

Gets the specified weather data at the specified hour and time step index within that hour

Parameters:
- **state** – An active EnergyPlus "state" that is returned from a call to *api.state_manager.new_state()*.
- **hour** – Integer hour of day (0 to 23)
- **time_step_number** – Time step index in hour, from 1 to the number of zone time steps per hour

Returns: Value of the weather condition at the specified time

`tomorrow_weather_horizontal_ir_at_time`(*state: c_void_p*, *hour: int*, *time_step_number: int*)→ float [source]

Gets the specified weather data at the specified hour and time step index within that hour

Parameters:
- **state** – An active EnergyPlus "state" that is returned from a call to *api.state_manager.new_state()*.
- **hour** – Integer hour of day (0 to 23)
- **time_step_number** – Time step index in hour, from 1 to the number of zone time steps per hour

Returns: Value of the weather condition at the specified time

`tomorrow_weather_is_raining_at_time`(*state: c_void_p*, *hour: int*, *time_step_number: int*)→ bool [source]

Gets the specified weather data at the specified hour and time step index within that hour

Parameters:
- **state** – An active EnergyPlus "state" that is returned from a call to *api.state_manager.new_state()*.
- **hour** – Integer hour of day (0 to 23)
- **time_step_number** – Time step index in hour, from 1 to the number of zone time steps per hour

Returns: A true/false for whether the weather condition is active at the specified time

`tomorrow_weather_is_snowing_at_time`(*state: c_void_p*, *hour: int*, *time_step_number: int*)→ bool [source]

Gets the specified weather data at the specified hour and time step index within that hour

Parameters:
- **state** – An active EnergyPlus "state" that is returned from a call to *api.state_manager.new_state()*.

- **hour** – Integer hour of day (0 to 23)
- **time_step_number** – Time step index in hour, from 1 to the number of zone time steps per hour

Returns: A true/false for whether the weather condition is active at the specified time

`tomorrow_weather_liquid_precipitation_at_time`(*state: c_void_p*, *hour: int*, *time_step_number: int*)→ float [source]

Gets the specified weather data at the specified hour and time step index within that hour

Parameters:
- **state** – An active EnergyPlus "state" that is returned from a call to *api.state_manager.new_state()*.
- **hour** – Integer hour of day (0 to 23)
- **time_step_number** – Time step index in hour, from 1 to the number of zone time steps per hour

Returns: Value of the weather condition at the specified time

`tomorrow_weather_outdoor_barometric_pressure_at_time`(*state: c_void_p*, *hour: int*, *time_step_number: int*)→ float [source]

Gets the specified weather data at the specified hour and time step index within that hour

Parameters:
- **state** – An active EnergyPlus "state" that is returned from a call to *api.state_manager.new_state()*.
- **hour** – Integer hour of day (0 to 23)
- **time_step_number** – Time step index in hour, from 1 to the number of zone time steps per hour

Returns: Value of the weather condition at the specified time

`tomorrow_weather_outdoor_dew_point_at_time`(*state: c_void_p*, *hour: int*, *time_step_number: int*)→ float [source]

Gets the specified weather data at the specified hour and time step index within that hour

Parameters:
- **state** – An active EnergyPlus "state" that is returned from a call to *api.state_manager.new_state()*.
- **hour** – Integer hour of day (0 to 23)
- **time_step_number** – Time step index in hour, from 1 to the number of zone time steps per hour

Returns: Value of the weather condition at the specified time

`tomorrow_weather_outdoor_dry_bulb_at_time`(*state: c_void_p*, *hour: int*, *time_step_number: int*)→ float [source]

Gets the specified weather data at the specified hour and time step index within that hour

Parameters:
- **state** – An active EnergyPlus "state" that is returned from a call to *api.state_manager.new_state()*.
- **hour** – Integer hour of day (0 to 23)
- **time_step_number** – Time step index in hour, from 1 to the number of zone time steps per hour

Returns: Value of the weather condition at the specified time

`tomorrow_weather_outdoor_relative_humidity_at_time`(*state: c_void_p*, *hour: int*, *time_step_number: int*)→ float [source]

Gets the specified weather data at the specified hour and time step index within that hour

Parameters:
- **state** – An active EnergyPlus "state" that is returned from a call to *api.state_manager.new_state()*.
- **hour** – Integer hour of day (0 to 23)
- **time_step_number** – Time step index in hour, from 1 to the number of zone time steps per hour

Returns: Value of the weather condition at the specified time

`tomorrow_weather_sky_temperature_at_time`(*state: c_void_p*, *hour: int*, *time_step_number: int*)→ float [source]

Gets the specified weather data at the specified hour and time step index within that hour

Parameters:
- **state** – An active EnergyPlus "state" that is returned from a call to *api.state_manager.new_state()*.
- **hour** – Integer hour of day (0 to 23)
- **time_step_number** – Time step index in hour, from 1 to the number of zone time steps per hour

Returns: Value of the weather condition at the specified time

`tomorrow_weather_wind_direction_at_time`(*state: c_void_p*, *hour: int*, *time_step_number: int*)→ float [source]

Gets the specified weather data at the specified hour and time step index within that hour

Parameters:
- **state** – An active EnergyPlus “state” that is returned from a call to *api.state_manager.new_state()*.
- **hour** – Integer hour of day (0 to 23)
- **time_step_number** – Time step index in hour, from 1 to the number of zone time steps per hour

Returns: Value of the weather condition at the specified time

`tomorrow_weather_wind_speed_at_time`(*state: c_void_p*, *hour: int*, *time_step_number: int*)→ float [source]

Gets the specified weather data at the specified hour and time step index within that hour

Parameters:
- **state** – An active EnergyPlus “state” that is returned from a call to *api.state_manager.new_state()*.
- **hour** – Integer hour of day (0 to 23)
- **time_step_number** – Time step index in hour, from 1 to the number of zone time steps per hour

Returns: Value of the weather condition at the specified time

`warmup_flag`(*state: c_void_p*)→ bool [source]

Gets a flag for whether the warmup flag is currently on, signaling that EnergyPlus is still in the process of converging on warmup days. The C API returns an integer where 1 is yes and 0 is no, this simply wraps that with a bool conversion.

Parameters: **state** – An active EnergyPlus “state” that is returned from a call to *api.state_manager.new_state()*.

Returns: A boolean indicating whether the warmup flag is on.

`year`(*state: c_void_p*)→ int [source]

Get the “current” year of the simulation, read from the EPW. All simulations operate at a real year, either user specified or automatically selected by EnergyPlus based on other data (start day of week + leap year option).

Parameters: **state** – An active EnergyPlus “state” that is returned from a call to *api.state_manager.new_state()*.

Returns: An integer year (2020, for example)

`zone_time_step`(*state: c_void_p*)→ float [source]

Gets the current zone time step value in EnergyPlus. The zone time step is variable and fluctuates during the simulation.

Parameters: **state** – An active EnergyPlus “state” that is returned from a call to *api.state_manager.new_state()*.

Returns: The current zone time step in fractional hours.

`zone_time_step_number`(*state: c_void_p*)→ int [source]

The current zone time step index, from 1 to the number of zone time steps per hour

Parameters: **state** – An active EnergyPlus “state” that is returned from a call to *api.state_manager.new_state()*.

Returns: The integer index of the current time step