ecFlow Python Api

        - set_child_try_no((Client)arg1, (int)arg2) None :
        - set_connection_attempts((Client)arg1, (int)arg2) None :
        - set_host_port((Client)arg1, (str)arg2, (str)arg3) None :
        - set_retry_connection_period((Client)arg1, (int)arg2) None :
        - shutdown_server((Client)arg1) int :
        - sort_attributes((Client)arg1, (str)abs_node_path, (str)attribute_name[, (bool)recursive=True]) None
        - stats((Client)arg1) None :
        - stats_reset((Client)arg1) None :
        - status((Client)arg1, (str)arg2) None :
        - suites((Client)arg1) list :
        - suspend((Client)arg1, (str)arg2) None :
        - sync_local((Client)arg1) int :
        - terminate_server((Client)arg1) int :
        - version((Client)arg1) str :
        - wait_for_server_reply((Client)arg1[, (int)time_out=60]) bool :
        - zombie_adopt((Client)arg1, (str)arg2) int
        - zombie_block((Client)arg1, (str)arg2) int
        - zombie_fail((Client)arg1, (str)arg2) int
        - zombie_fob((Client)arg1, (str)arg2) int
        - zombie_kill((Client)arg1, (str)arg2) int
        - zombie_remove((Client)arg1, (str)arg2) int
    - class ecflow.Clock
        - day((Clock)arg1) int :
        - gain((Clock)arg1) int :
        - month((Clock)arg1) int :
        - positive_gain((Clock)arg1) bool :
        - set_gain((Clock)arg1, (int)arg2, (int)arg3, (bool)arg4) None :
        - set_gain_in_seconds((Clock)arg1, (int)arg2, (bool)arg3) None :
        - set_virtual((Clock)arg1, (bool)arg2) None :
        - virtual((Clock)arg1) bool :
        - year((Clock)arg1) int :
    - class ecflow.Complete
        - get_expression((Complete)arg1) str :
    - class ecflow.Cron
        - set_days_of_month((Cron)arg1, (list)arg2) None :
        - set_months((Cron)arg1, (list)arg2) None :
        - set_time_series((Cron)arg1, (int)hour, (int)minute[, (bool)relative=False]) None :
        - set_week_days((Cron)arg1, (list)arg2) None :
        - time((Cron)arg1) TimeSeries :
    - class ecflow.DState
    - class ecflow.Date
        - day((Date)arg1) int :
        - month((Date)arg1) int :
        - year((Date)arg1) int :
    - class ecflow.Day
        - day((Day)arg1) Days :
    - class ecflow.Days
    - class ecflow.Defs
        - add()
        - add_extern((Defs)arg1, (str)arg2) None :
        - add_suite((Defs)arg1, (Suite)arg2) Suite :
        - add_variable((Defs)arg1, (str)arg2, (str)arg3) Defs :
        - auto_add_externs((Defs)arg1, (bool)arg2) None :
        - check((Defs)arg1) str :
        - check_job_creation((Defs)arg1[, (bool)throw_on_error=False[, (bool)verbose=False]]) str :
        - delete_variable((Defs)arg1, (str)arg2) None :
        - find_abs_node((Defs)arg1, (str)arg2) Node :
        - find_suite((Defs)arg1, (str)arg2) Suite :
        - generate_scripts((Defs)arg1) None :
        - get_all_nodes((Defs)arg1) NodeVec :
        - get_all_tasks((Defs)arg1) TaskVec :
        - get_server_state((Defs)arg1) SState :
        - get_state((Defs)arg1) State
        - has_time_dependencies((Defs)arg1) bool :
        - restore_from_checkpt((Defs)arg1, (str)arg2) None :
        - save_as_checkpt((Defs)arg1, (str)arg2) None :
        - save_as_defs((Defs)arg1, (str)arg2[, (Style)arg3]) None :
        - simulate((Defs)arg1) str :
        - sort_attributes((Defs)arg1, (str)attribute_type[, (bool)recursive=True]) None
    - class ecflow.Defstatus
        - state((Defstatus)arg1) DState
    - class ecflow.Ecf
        - static debug_equality() bool :
        - static debug_level() int :
        - static set_debug_equality((bool)arg1) None :
        - static set_debug_level((int)arg1) None :
    - class ecflow.Edit
    - class ecflow.Event
        - empty((Event)arg1) bool :

        - name((Event)arg1) str :
        - name_or_number((Event)arg1) str :
        - number((Event)arg1) int :
        - value((Event)arg1) bool :
    - class ecflow.Expression
        - add((Expression)arg1, (PartExpression)arg2) None :
        - get_expression((Expression)arg1) str :
    - class ecflow.Family
    - class ecflow.FamilyVec
        - append((FamilyVec)arg1, (object)arg2) None
        - extend((FamilyVec)arg1, (object)arg2) None
    - class ecflow.File
        - static build_dir() str :
        - static find_client() str :
        - static find_server() str :
        - static source_dir() str :
    - class ecflow.Flag
        - clear((Flag)arg1, (FlagType)arg2) None :
        - is_set((Flag)arg1, (FlagType)arg2) bool :
        - static list() FlagTypeVec :
        - reset((Flag)arg1) None :
        - set((Flag)arg1, (FlagType)arg2) None :
        - static type_to_string((FlagType)arg1) str :
    - class ecflow.FlagType
    - class ecflow.FlagTypeVec
        - append((FlagTypeVec)arg1, (object)arg2) None
        - extend((FlagTypeVec)arg1, (object)arg2) None
    - class ecflow.InLimit
        - name((InLimit)arg1) str :
        - path_to_node((InLimit)arg1) str :
        - tokens((InLimit)arg1) int :
    - class ecflow.JobCreationCtrl
        - generate_temp_dir((JobCreationCtrl)arg1) None :
        - get_dir_for_job_creation((JobCreationCtrl)arg1) str :
        - get_error_msg((JobCreationCtrl)arg1) str :
        - set_dir_for_job_creation((JobCreationCtrl)arg1, (str)arg2) None :
        - set_node_path((JobCreationCtrl)arg1, (str)arg2) None :
        - set_verbose((JobCreationCtrl)arg1, (bool)arg2) None :
    - class ecflow.Label
        - empty((Label)arg1) bool :
        - name((Label)arg1) str :
        - new_value((Label)arg1) str :
        - value((Label)arg1) str :
    - class ecflow.Late
        - active((Late)arg1, (int)arg2, (int)arg3) None :
        - complete((Late)arg1, (int)arg2, (int)arg3, (bool)arg4) None :
        - complete_is_relative((Late)arg1) bool :
        - is_late((Late)arg1) bool :
        - submitted((Late)arg1, (TimeSlot)arg2) None :
    - class ecflow.Limit
        - decrement((Limit)arg1, (int)arg2, (str)arg3) None :
        - increment((Limit)arg1, (int)arg2, (str)arg3) None :
        - limit((Limit)arg1) int :
        - name((Limit)arg1) str :
        - node_paths((Limit)arg1) list :
        - value((Limit)arg1) int :
    - class ecflow.Meter
        - color_change((Meter)arg1) int :
        - empty((Meter)arg1) bool :
        - max((Meter)arg1) int :
        - min((Meter)arg1) int :
        - name((Meter)arg1) str :
        - value((Meter)arg1) int :
    - class ecflow.Node
        - add()
        - add_autocancel((Node)arg1, (int)arg2) Node :
        - add_complete((Node)arg1, (str)arg2) Node :
        - add_cron((Node)arg1, (Cron)arg2) Node :
        - add_date((Node)arg1, (int)arg2, (int)arg3, (int)arg4) Node :
        - add_day((Node)arg1, (Days)arg2) Node :
        - add_defstatus((Node)arg1, (DState)arg2) Node :
        - add_event((Node)arg1, (Event)arg2) Node :
        - add_inlimit((Node)arg1, (str)limit_name[, (str)path_to_node_containing_limit=''[, (int)tokens=1]]) Node :
        - add_label((Node)arg1, (str)arg2, (str)arg3) Node :
        - add_late((Node)arg1, (Late)arg2) Node :
        - add_limit((Node)arg1, (str)arg2, (int)arg3) Node :
        - add_meter((Node)arg1, (Meter)arg2) Node :
        - add_part_complete((Node)arg1, (PartExpression)arg2) Node :
        - add_part_trigger((Node)arg1, (PartExpression)arg2) Node :

        - add_repeat((Node)arg1, (RepeatDate)arg2) Node :
        - add_time((Node)arg1, (int)arg2, (int)arg3) Node :
        - add_today((Node)arg1, (int)arg2, (int)arg3) Node :
        - add_trigger((Node)arg1, (str)arg2) Node :
        - add_variable((Node)arg1, (str)arg2, (str)arg3) Node :
        - add_verify((Node)arg1, (Verify)arg2) None :
        - add_zombie((Node)arg1, (ZombieAttr)arg2) Node :
        - change_complete((Node)arg1, (str)arg2) None
        - change_trigger((Node)arg1, (str)arg2) None
        - delete_complete((Node)arg1) None
        - delete_cron((Node)arg1, (str)arg2) None
        - delete_date((Node)arg1, (str)arg2) None
        - delete_day((Node)arg1, (str)arg2) None
        - delete_event((Node)arg1, (str)arg2) None
        - delete_inlimit((Node)arg1, (str)arg2) None
        - delete_label((Node)arg1, (str)arg2) None
        - delete_limit((Node)arg1, (str)arg2) None
        - delete_meter((Node)arg1, (str)arg2) None
        - delete_repeat((Node)arg1) None
        - delete_time((Node)arg1, (str)arg2) None
        - delete_today((Node)arg1, (str)arg2) None
        - delete_trigger((Node)arg1) None
        - delete_variable((Node)arg1, (str)arg2) None
        - delete_zombie((Node)arg1, (str)arg2) None
        - evaluate_complete((Node)arg1) bool :
        - evaluate_trigger((Node)arg1) bool :
        - find_event((Node)arg1, (str)arg2) Event :
        - find_gen_variable((Node)arg1, (str)arg2) Variable :
        - find_label((Node)arg1, (str)arg2) Label :
        - find_limit((Node)arg1, (str)arg2) Limit :
        - find_meter((Node)arg1, (str)arg2) Meter :
        - find_node_up_the_tree((Node)arg1, (str)arg2) Node :
        - find_parent_variable((Node)arg1, (str)arg2) Variable :
        - find_variable((Node)arg1, (str)arg2) Variable :
        - get_abs_node_path((Node)arg1) str :
        - get_all_nodes((Node)arg1) NodeVec :
        - get_autocancel((Node)arg1) Autocancel
        - get_complete((Node)arg1) Expression
        - get_defs((Node)arg1) Defs
        - get_defstatus((Node)arg1) DState
        - get_dstate((Node)arg1) DState :
        - get_flag((Node)arg1) Flag :
        - get_generated_variables((Node)arg1, (VariableList)arg2) None :
        - get_late((Node)arg1) Late
        - get_parent((Node)arg1) Node
        - get_repeat((Node)arg1) Repeat
        - get_state((Node)arg1) State :
        - get_state_change_time((Node)arg1[, (str)format='iso_extended']) str :
        - get_trigger((Node)arg1) Expression
        - has_time_dependencies((Node)arg1) bool
        - is_suspended((Node)arg1) bool :
        - name((Node)arg1) str
        - remove((Node)arg1) Node :
        - replace_on_server((Node)arg1[, (bool)suspend_node_first=True[, (bool)force=True]]) None :
        - sort_attributes((Node)arg1, (AttrType)attribute_type[, (bool)recursive=True]) None
        - update_generated_variables((Node)arg1) None
    - class ecflow.NodeContainer
        - add_family((NodeContainer)arg1, (str)arg2) Family :
        - add_task((NodeContainer)arg1, (str)arg2) Task :
        - find_family((NodeContainer)arg1, (str)arg2) Family :
        - find_task((NodeContainer)arg1, (str)arg2) Task :
    - class ecflow.NodeVec
        - append((NodeVec)arg1, (object)arg2) None
        - extend((NodeVec)arg1, (object)arg2) None
    - class ecflow.PartExpression
        - and_expr((PartExpression)arg1) bool
        - get_expression((PartExpression)arg1) str :
        - or_expr((PartExpression)arg1) bool
    - class ecflow.PrintStyle
        - static get_style() Style :
        - static set_style((Style)arg1) None :
    - class ecflow.Repeat
        - empty((Repeat)arg1) bool :
        - end((Repeat)arg1) int :
        - name((Repeat)arg1) str :
        - start((Repeat)arg1) int :
        - step((Repeat)arg1) int :
        - value((Repeat)arg1) int :
    - class ecflow.RepeatDate

        - end((RepeatDate)arg1) int :
        - name((RepeatDate)arg1) str :
        - start((RepeatDate)arg1) int :
        - step((RepeatDate)arg1) int :
    - class ecflow.RepeatDay
    - class ecflow.RepeatEnumerated
        - end((RepeatEnumerated)arg1) int
        - name((RepeatEnumerated)arg1) str :
        - start((RepeatEnumerated)arg1) int
        - step((RepeatEnumerated)arg1) int
    - class ecflow.RepeatInteger
        - end((RepeatInteger)arg1) int
        - name((RepeatInteger)arg1) str :
        - start((RepeatInteger)arg1) int
        - step((RepeatInteger)arg1) int
    - class ecflow.RepeatString
        - end((RepeatString)arg1) int
        - name((RepeatString)arg1) str :
        - start((RepeatString)arg1) int
        - step((RepeatString)arg1) int
    - class ecflow.SState
    - class ecflow.State
    - class ecflow.Style
    - class ecflow.Submittable
        - get_aborted_reason((Submittable)arg1) str :
        - get_int_try_no((Submittable)arg1) int :
        - get_jobs_password((Submittable)arg1) str :
        - get_process_or_remote_id((Submittable)arg1) str :
        - get_try_no((Submittable)arg1) str :
    - class ecflow.Suite
        - add_clock((Suite)arg1, (Clock)arg2) Suite
        - add_end_clock((Suite)arg1, (Clock)arg2) Suite :
        - begun((Suite)arg1) bool :
        - get_clock((Suite)arg1) Clock :
        - get_end_clock((Suite)arg1) Clock :
    - class ecflow.SuiteVec
        - append((SuiteVec)arg1, (object)arg2) None
        - extend((SuiteVec)arg1, (object)arg2) None
    - class ecflow.Task
    - class ecflow.TaskVec
        - append((TaskVec)arg1, (object)arg2) None
        - extend((TaskVec)arg1, (object)arg2) None
    - class ecflow.Time
        - time_series((Time)arg1) TimeSeries :
    - class ecflow.TimeSeries
        - finish((TimeSeries)arg1) TimeSlot :
        - has_increment((TimeSeries)arg1) bool :
        - incr((TimeSeries)arg1) TimeSlot :
        - relative((TimeSeries)arg1) bool :
        - start((TimeSeries)arg1) TimeSlot :
    - class ecflow.TimeSlot
        - empty((TimeSlot)arg1) bool
        - hour((TimeSlot)arg1) int
        - minute((TimeSlot)arg1) int
    - class ecflow.Today
        - time_series((Today)arg1) TimeSeries :
    - class ecflow.Trigger
        - get_expression((Trigger)arg1) str :
    - class ecflow.UrlCmd
        - execute((UrlCmd)arg1) None :
    - class ecflow.Variable
        - empty((Variable)arg1) bool :
        - name((Variable)arg1) str :
        - value((Variable)arg1) str :
    - class ecflow.VariableList
        - append((VariableList)arg1, (object)arg2) None
        - extend((VariableList)arg1, (object)arg2) None
    - class ecflow.Verify
    - class ecflow.WhyCmd
        - why((WhyCmd)arg1) str :
    - class ecflow.ZombieAttr
        - empty((ZombieAttr)arg1) bool :
        - user_action((ZombieAttr)arg1) ZombieUserActionType :
        - zombie_lifetime((ZombieAttr)arg1) int :
        - zombie_type((ZombieAttr)arg1) ZombieType :
    - class ecflow.ZombieType
    - class ecflow.ZombieUserActionType

ecFlow Python Api

Api

The ecflow module provides the python bindings/api for creating definition structure and communicating with the server.

class ecflow.Alias

Bases: `ecflow.Submittable`

A Aliases is create by the GUI or via edit_script command

Aliases provide a mechanism to edit/test task scripts without effecting the suite The Aliases parent is always a Task.Multiple Alias can be added

class ecflow.AttrType

Bases: `Boost.Python.enum`

Sortable attribute type, currently [event | meter | label | limit | variable]

```
event = ecflow.AttrType.event
label = ecflow.AttrType.label
limit = ecflow.AttrType.limit
meter = ecflow.AttrType.meter
names = {'variable': ecflow.AttrType.variable, 'limit': ecflow.AttrType.limit, 'event': ecflow.AttrType.event,
'meter': ecflow.AttrType.meter, 'label': ecflow.AttrType.label}
values = {1: ecflow.AttrType.event, 2: ecflow.AttrType.meter, 3: ecflow.AttrType.label, 4: ecflow.AttrType.
limit, 5: ecflow.AttrType.variable}
variable = ecflow.AttrType.variable
```

class ecflow.Autocancel

Bases: `Boost.Python.instance`

Provides a way to automatically delete/remove a node which has completed

See autocancel

Constructor:

```
Autocancel(TimeSlot,relative)
   TimeSlot single: A time
   bool relative:   Relative to completion. False means delete the node at the real time specified.

Autocancel(hour,minute,relative)
   int hour:        hour in 24 hrs
   int minute:      minute &lt;= 59
   bool relative:   Relative to completion. False means delete the node at the real time specified.

Autocancel(days)
   int days:        Delete the node "days" after completion
```

Usage:

```
attr = Autocancel( 1,30, true )             # delete node 1 hour and 30 minutes after completion
attr = Autocancel( TimeSlot(0,10), true )    # delete node 10 minutes after completion
attr = Autocancel( TimeSlot(10,10), false )  # delete node at 10:10 after completion
attr = Autocancel( 3  )                      # delete node 3 days after completion

t1 = Task("t1",
          Autocancel(2,0,true))              # delete task 2 hours after completion
```

days((Autocancel)arg1) bool :

Returns a boolean true if time was specified in days

relative((Autocancel)arg1) bool :

Returns a boolean where true means the time is relative

time((Autocancel)arg1) TimeSlot :

returns cancel time as a TimeSlot

class ecflow.CheckPt

Bases: `Boost.Python.enum`

CheckPt is enum that is used to control check pointing in the ecflow_server

- NEVER : Switches of check pointing
- ON_TIME: check point file is saved periodically, specified by checkPtInterval. This is the default.
- ALWAYS : check point file is saved after any state change, *not* recommended for large definitions
- UNDEFINED : None of the the above, used to provide default argument

```
ALWAYS = ecflow.CheckPt.ALWAYS
NEVER = ecflow.CheckPt.NEVER
ON_TIME = ecflow.CheckPt.ON_TIME
UNDEFINED = ecflow.CheckPt.UNDEFINED
names = {'ALWAYS': ecflow.CheckPt.ALWAYS, 'NEVER': ecflow.CheckPt.NEVER, 'UNDEFINED': ecflow.CheckPt.UNDEFINED,
'ON_TIME': ecflow.CheckPt.ON_TIME}
values = {0: ecflow.CheckPt.NEVER, 1: ecflow.CheckPt.ON_TIME, 2: ecflow.CheckPt.ALWAYS, 3: ecflow.CheckPt.
UNDEFINED}
```

class ecflow.ChildCmdType

Bases: `Boost.Python.enum`

ChildCmdType represents the different child command s. This type is used as a parameter to the class `ecflow.ZombieAttr`

Child commands are called within a job file:

```
ChildCmdType::init     corresponds to : ecflow_client --init=&lt;process_id&gt;
ChildCmdType::event    corresponds to : ecflow_client --event=&lt;event_name | number&gt;
ChildCmdType::meter    corresponds to : ecflow_client --meter=&lt;meter_name&gt;, &lt;meter_value&gt;
ChildCmdType::label    corresponds to : ecflow_client --label=&lt;label_name&gt;. &lt;label_value&gt;
ChildCmdType::wait     corresponds to : ecflow_client --wait=&lt;expression&gt;
ChildCmdType::abort    corresponds to : ecflow_client --abort=&lt;reason&gt;
ChildCmdType::complete corresponds to : ecflow_client --complete
```

```
abort = ecflow.ChildCmdType.abort
complete = ecflow.ChildCmdType.complete
event = ecflow.ChildCmdType.event
init = ecflow.ChildCmdType.init
label = ecflow.ChildCmdType.label
meter = ecflow.ChildCmdType.meter
names = {'complete': ecflow.ChildCmdType.complete, 'meter': ecflow.ChildCmdType.meter, 'label': ecflow.
ChildCmdType.label, 'init': ecflow.ChildCmdType.init, 'abort': ecflow.ChildCmdType.abort, 'event': ecflow.
ChildCmdType.event, 'wait': ecflow.ChildCmdType.wait}
values = {0: ecflow.ChildCmdType.init, 1: ecflow.ChildCmdType.event, 2: ecflow.ChildCmdType.meter, 3: ecflow.
ChildCmdType.label, 4: ecflow.ChildCmdType.wait, 5: ecflow.ChildCmdType.abort, 6: ecflow.ChildCmdType.complete}
wait = ecflow.ChildCmdType.wait
```

class ecflow.Client

Bases: `Boost.Python.instance`

Class client provides an interface to communicate with the ecflow_server.:

```
Client(
   string host, # The server name. Can not be empty.
   string port  # The port on the server, must be unique to the server
)

Client(
   string host, # The server name. Can not be empty.
   int port     # The port on the server, must be unique to the server
)

Client(
   string host_port, # Expect"s &lt;host&gt;:&lt;port&gt; || &lt;host&gt;@&lt;port&gt;
)
```

The client reads in the following environment variables. For child commands,(i.e. these are commands called in the .ecf/jobs files), these variables are used. For the python interface these environment variable are not really applicable but documented for completeness:

- ECF_NAME <string> : Full path name to the task
- ECF_PASS <string> : The jobs password, allocated by server, then used by server to authenticate client request
- ECF_TRYNO <int> : The number of times the job has run. Used in file name generation. Set to 1 by begin() and re-queue commands.
- ECF_TIMEOUT <int> : Max time in seconds for client to deliver message to main server
- ECF_HOSTFILE <string> : File that lists alternate hosts to try, if connection to main host fails
- ECF_DENIED <any> : Provides a way for child to exit with an error, if server denies connection. Avoids 24hr wait. Note: when you have hundreds of tasks, using this approach requires a lot of manual intervention to determine job status
- NO_ECF <any> : If set exit's immediately with success. Used to test jobs without communicating with server

The following environment variables are used by the python interface and child commands

- ECF_HOST <string> : The host name of the main server. defaults to 'localhost'
- ECF_PORT <int> : The TCP/IP port to call on the server. Must be unique to a server

The ECF_HOST and ECF_PORT can be overridden by using the Constructor or set_host_port() member function. For optimal usage it is best to reuse the same Client rather than recreating for each client server interaction By default the Client interface will throw exceptions for error's.

Usage:

```
try:
    ci = Client("localhost:3150")   # for errors will throw RuntimeError
    ci.terminate_server()
except RuntimeError, e:
    print(str(e))
```

alter((Client)arg1, (list)paths, (str)alter_type, (str)attribute_type[, (str)name=''[, (str)value='']]) None :

Alter command is used to change the attributes of a node

```
void alter(
   (list | string ) paths(s) : A single or list of paths. Path name to the node whose
attributes are to be changed
   string alter_type         : This must be one of [ "add" | "change" | "delete" |
"set_flag" | "clear_flag" ]
   string attr_type          : This varies according to the "alter_type". valid strings are:
      add    : [ variable,time,today,date,day,label,zombie,late]
      delete : [ variable,time,today,date,day,label,cron,event,meter,trigger,complete,
repeat,limit,inlimit,limit_path,zombie,late]
      change : [ variable,clock-type,clock-gain,event,meter,label,trigger,complete,repeat,
limit-max,limit-value,late]
      set_flag and clear_flag:
               [ force_aborted | user_edit | task_aborted | edit_failed | ecfcmd_failed |
no_script | killed |
                 migrated | late | message | complete | queue_limit | task_waiting | locked
| zombie ]
   string name               : used to locate the attribute, when multiple attributes of
the same type,
                               optional for some.i.e.. when changing, attributes like
variable,meter,event,label,limits
   string value              : Only used when "changing" a attribute. provides a new value
)
```

Exceptions can be raised because:

- absolute_node_path does not exist.
- parsing fails

The following describes the parameters in more detail:

```
add variable variable_name variable_value
add time   format    # when format is +hh:mm | hh:mm | hh:mm(start) hh:mm(finish) hh:mm
(increment)
add today  format    # when format is +hh:mm | hh:mm | hh:mm(start) hh:mm(finish) hh:mm
(increment)
add date   format    # when format dd.mm.yyyy, can use "*" to indicate any day,month, or
year
add day    format    # when format is one of [ sunday,monday,tuesday,wednesday,friday,
saturday ]
add zombie format    # when format is one of &lt;zombie-type&gt;:&lt;child&gt;:&lt;server-
action&gt;|&lt;client-action&gt;:&lt;zombie-lifetime&gt;
                     #  &lt;zombie-type&gt; := [ user | ecf | path ]
                     #  &lt;child&gt; := [ init, event, meter, label, wait, abort, complete
]
                     #  &lt;server-action&gt; := [ adopt | delete ]
                     #  &lt;client-action&gt; := [ fob | fail | block(default) ]
                     #  &lt;zombie-lifetime&gt;:= lifetime of zombie in the server
                     # example
                     # add zombie :label:fob:0   # fob all child label request, &amp;
remove zombie as soon as possible

delete variable name # if name is empty will delete -all- variables on the node
delete time name     # To delete a specific time, enter the time in same format as show
above,
                     # or as specified in the defs file
                     # an empty name will delete all time attributes on the node
delete today name    # To delete a specific today attribute, enter in same format as show
above,
                     # or as specified in the defs file.
                     # an empty name will delete all today attributes on the node
delete date name     # To delete a specific date attribute, enter in same format as show
above,
                     # or as specified in the defs file
                     # an empty name will delete all date attributes on the node
delete day name      # To delete a specific day attribute, enter in same format as show
above,
                     # or as specified in the defs file
                     # an empty name will delete all day attributes on the node
delete cron name     # To delete a specific cron attribute, enter in same as specified in
```

```
the defs file
                    # an empty name will delete all cron attributes on the node
delete event name   # To delete a specific event, enter name or number
                    # an empty name will delete all events on the node
delete meter name   # To delete a specific meter , enter the meter name
                    # an empty name will delete all meter on the node
delete label name   # To delete a specific label , enter the label name
                    # an empty name will delete all labels on the node
delete limit name   # To delete a specific limit , enter the limit name
                    # an empty name will delete all limits on the node
delete inlimit name  # To delete a specific inlimit , enter the inlimit name
                    # an empty name will delete all inlimits on the node
delete limit_path limit_name limit_path # To delete a specific limit path
delete trigger      # A node can only have one trigger expression, hence the name is not
required
delete complete     # A node can only have one complete expression, hence the name is not
required
delete repeat       # A node can only have one repeat, hence the name is not required

change variable name value    # Find the specified variable, and set the new value.
change clock_type name        # The name must be one of "hybrid" or "real".
change clock_gain name        # The gain must be convertible to an integer.
change clock_sync name        # Sync suite calendar with the computer.
change event name(optional )  # if no name specified the event is set, otherwise name must
be "set" or "clear"
change meter name value       # The meter value must be convertible to an integer, and
between meter min-max range.
change label name value       # sets the label
change trigger name           # The name must be expression. returns an error if the
expression does not parse
change complete name          # The name must be expression. returns an error if the
expression does not parse
change limit_max name value   # Sets the max value of the limit. The value must be
convertible to an integer
change limit_value name value # Sets the consumed tokens to value. The value must be
convertible to an integer
change repeat value           # If the repeat is a date, then the value must be a valid YMD
( ie. yyyymmdd)
                              # and be convertible to an integer, additionally the value
must be in range
                              # of the repeat start and end dates. Like wise for repeat
integer. For repeat
                              # string and enum,  the name must either be an integer, that
is a valid index or
                              # if it is a string, it must correspond to one of enum"s or
strings list
```

Usage:

```
try:
   ci = Client()     # use default host(ECF_HOST) &amp;amp; port(ECF_PORT)
   ci.alter("/suite/task","change","trigger","b2 == complete")
except RuntimeError, e:
   print(str(e))
```

alter((Client)arg1, (str)abs_node_path, (str)alter_type, (str)attribute_type [, (str)name='' [, (str)value='']]) -> None

begin_all_suites((Client)arg1[, (bool)force=False]) int :

Begin playing all the suite s in the ecflow_server

Note

using the force option may cause zombie s if suite has running jobs

```
void begin_all_suites(
   [(bool)force=False] : bypass the checks for submitted and active jobs
)
```

Usage:

```
try:
    ci = Client()             # use default host(ECF_HOST) &amp;amp; port(ECF_PORT)
    ci.begin_all_suites()     # begin playing all the suites
    ci.begin_all_suites(True) # begin playing all the suites, by passing checks
except RuntimeError, e:
    print(str(e))
```

begin_suite((Client)arg1, (str)suite_name[, (bool)force=False]) int :

Begin playing the chosen suite s in the ecflow_server

Note

using the force option may cause zombie s if suite has running jobs

```
void begin_suite
   string suite_name     : begin playing the given suite
   [(bool)force=False]   : bypass the checks for submitted and active jobs
)
```

Usage:

```
try:
    ci = Client()                  # use default host(ECF_HOST) &amp;amp; port(ECF_PORT)
    ci.begin_suite("/suite1")      # begin playing suite "/suite1"
    ci.begin_suite("/suite1",True) # begin playing suite "/suite1" bypass any checks   except
RuntimeError, e:
    print(str(e))
```

ch_add((Client)arg1, (int)arg2, (list)arg3) None :

Add a set of suites, to an existing registered handle

When dealing with large definitions, where a user is only interested in a small subset of suites, registering them, improves download performance from the server. Registered suites have an associated handle.

```
integer ch_add(
   integer handle   : the handle obtained after ch_register
   list suite_names : list of strings representing suite names
)
integer ch_add(
   list suite_names : list of strings representing suite names
)
```

Usage:

```
try:
    ci = Client()           # use default host(ECF_HOST) &amp;amp; port(ECF_PORT)
    ci.ch_register(True,[]) # register interest in any new suites
    ci.ch_add(["s1","s2"])  # add suites s1,s2 to the last added handle
except RuntimeError, e:
    print(str(e))
```

ch_add((Client)arg1, (list)arg2) -> None

ch_auto_add((Client)arg1, (int)arg2, (bool)arg3) int :

Change an existing handle so that new suites can be added automatically

When dealing with large definitions, where a user is only interested in a small subset of suites, registering them, improves download performance from the server. Registered suites have an associated handle.

```
void ch_auto_add(
   integer handle,         : the handle obtained after ch_register
   bool auto_add_new_suite : automatically add new suites, this handle when they are created
)
void ch_auto_add(
   bool auto_add_new_suite : automatically add new suites using handle on the client
)
```

Usage:

```
try:
    ci = Client()                         # use default host(ECF_HOST) &amp; port(ECF_PORT)
    ci.ch_register(True,["s1","s2","s3"]) # register interest in suites s1,s2,s3 and any
new suites
    ci.ch_auto_add( False )               # disable adding newly created suites to my handle
except RuntimeError, e:
    print(str(e))
```

ch_auto_add((Client)arg1, (bool)arg2) -> int

ch_drop((Client)arg1, (int)arg2) int :

Drop/de-register the client handle.

When dealing with large definitions, where a user is only interested in a small subset of suites, registering them, improves download performance from the server. Registered suites have an associated handle. Client must ensure un-used handle are dropped otherwise they will stay, in the ecflow_server

```
void ch_drop(
   int client_handle : The handle must be an integer that is &gt; 0
)
void ch_drop()       : Uses the local handle stored on the client, from last call to ch_register()
```

Exception:

- RunTimeError thrown if handle has not been previously registered

Usage:

```
try:
  ci = Client()                    # use default host(ECF_HOST) &amp; port(ECF_PORT)
  ci.ch_register(False,["s1","s2"])
  while( 1 ):
     # get incremental changes to suites s1 &amp; s2, uses data stored on ci/defs
     ci.sync_local()               # will only retrieve data for suites s1 &amp; s2
     update(ci.get_defs())
finally:
  ci.ch_drop()
```

ch_drop((Client)arg1) -> int

ch_drop_user((Client)arg1, (str)arg2) int :

Drop/de-register all handles associated with user.

When dealing with large definitions, where a user is only interested in a small subset of suites, registering them, improves download performance from the server. Registered suites have an associated handle. Client must ensure un-used handle are dropped otherwise they will stay, in the ecflow_server

```
void ch_drop_user(
     string user   # If empty string will drop current user
)
```

Exception:

- RunTimeError thrown if handle has not been previously registered

Usage:

```
try:
    ci = Client()                     # use default host(ECF_HOST) &amp;amp; port(ECF_PORT)
    ci.ch_register(False,["s1","s2"])
    while( 1 ):
       # get incremental changes to suites s1 &amp;amp; s2, uses data stored on ci/defs
       update(ci.get_defs())
finally:
    ci.ch_drop_user("") # drop all handles associated with current user
```

ch_handle((Client)arg1) int :

Register interest in a set of suite s.

If a definition has lots of suites, but the client is only interested in a small subset. Then using this command can reduce network bandwidth and synchronisation will be quicker. This command will create a client handle. This handle is held locally on the `ecflow.Client`, and can be used implicitly by ch_drop(),ch_add(),ch_remove() and ch_auto_add(). Registering a client handle affects the news() and sync() commands:

```
void ch_register(
   bool auto_add_new_suites : true means add new suites to my list, when they are created
   list suite_names         : should be a list of suite names, names not in the definition are
ignored
)
```

Usage:

```
try:
    ci = Client()
    suite_names = [ "s1", "s2", "s3" ]
    ci.ch_register(True,suite_names)    # register interest in suites s1,s2,s3 and any new suites
    ci.ch_register(False,suite_names)   # register interest in suites s1,s2,s3 only
except RuntimeError, e:
    print(str(e))
```

The client ‘ci’ will hold locally the client handle. Since we have made multiple calls to register a handle, the variable ‘ci’ will hold the handle for the last call only. The handle associated with the suite can be manually retrieved:

```
try:
    ci = Client()
    ci.ch_register(True,["s1","s2","s3"]) # register interest in suites s1,s2,s3 and any new suites
    client_handle = ci.ch_handle()        # get the handle associated with last call to ch_register
    ....                                  # after a period of time
except RuntimeError, e:
    print(str(e))
finally:
    ci.ch_drop( client_handle )           # de-register the handle
```

ch_register((Client)arg1, (bool)arg2, (list)arg3) None :

Register interest in a set of suite s.

If a definition has lots of suites, but the client is only interested in a small subset. Then using this command can reduce network bandwidth and synchronisation will be quicker. This command will create a client handle. This handle is held locally on the `ecflow.Client`, and can be used implicitly by ch_drop(),ch_add(),ch_remove() and ch_auto_add(). Registering a client handle affects the news() and sync() commands:

```
void ch_register(
   bool auto_add_new_suites : true means add new suites to my list, when they are created
   list suite_names         : should be a list of suite names, names not in the definition are
ignored
)
```

Usage:

```
try:
    ci = Client()
    suite_names = [ "s1", "s2", "s3" ]
    ci.ch_register(True,suite_names)    # register interest in suites s1,s2,s3 and any new suites
    ci.ch_register(False,suite_names)   # register interest in suites s1,s2,s3 only
except RuntimeError, e:
    print(str(e))
```

The client 'ci' will hold locally the client handle. Since we have made multiple calls to register a handle, the variable 'ci' will hold the handle for the last call only. The handle associated with the suite can be manually retrieved:

```
try:
    ci = Client()
    ci.ch_register(True,["s1","s2","s3"]) # register interest in suites s1,s2,s3 and any new suites
    client_handle = ci.ch_handle()        # get the handle associated with last call to ch_register
    ....                                  # after a period of time
except RuntimeError, e:
    print(str(e))
finally:
    ci.ch_drop( client_handle )           # de-register the handle
```

ch_remove((Client)arg1, (int)arg2, (list)arg3) None :

Remove a set of suites, from an existing handle

When dealing with large definitions, where a user is only interested in a small subset of suites, registering them, improves download performance from the server. Registered suites have an associated handle.

```
integer ch_remove(
   integer handle   : the handle obtained after ch_register
   list suite_names : list of strings representing suite names
)
integer ch_remove(
   list suite_names : list of strings representing suite names
)
```

Usage:

```
try:
    ci = Client()                         # use default host(ECF_HOST) &amp;amp; port(ECF_PORT)
    ci.ch_register(True,["s1","s2","s3"]) # register interest in suites s1,s2,s3 and any
new suites
    ci.ch_remove( ["s1"] )                # remove suites s1 from the last added handle
except RuntimeError, e:
    print(str(e))
```

ch_remove((Client)arg1, (list)arg2) -> None

ch_suites((Client)arg1) None :

Writes to standard out the list of registered handles and the suites they reference.

When dealing with large definitions, where a user is only interested in a small subset of suites, registering them, improves download performance from the server. Registered suites have an associated handle.

check((Client)arg1, (str)arg2) str :

Check trigger and complete expression s and limit s

The ecflow_server does not store extern s. Hence all unresolved references are reported as errors. Returns a non empty string for any errors or warning

```
string check(
   list paths # List of paths.
)
string check(
   string absolute_node_path
)
```

Usage:

```
try:
    ci = Client()   # use default host(ECF_HOST) &amp;amp; port(ECF_PORT)
    print(ci.check("/suite1"))
except RuntimeError, e:
    print(str(e))
```

check((Client)arg1, (list)arg2) -> str

checkpt((Client)arg1[, (CheckPt)mode=ecflow.CheckPt.UNDEFINED[, (int)check_pt_interval=0[, (int)check_pt_save_alarm_time=0]]]) int :

Request the ecflow_server check point s the definition held in the server immediately

This effectively saves the definition held in the server to disk, in a platform independent manner. This is the default when no arguments are specified. The saved file will include node state, passwords, etc. The default file name is <host>.<port>.ecf.check and is saved in ECF_HOME directory. The check point file name can be overridden via ECF_CHECK server environment variable. The back up check point file name can be overridden via ECF_CHECKOLD server environment variable:

```
void checkpt(
  [(CheckPt::Mode)mode=CheckPt.UNDEFINED]
                      : Must be one of [ NEVER, ON_TIME, ALWAYS, UNDEFINED ]
                        NEVER  :  Never check point the definition in the server
                        ON_TIME:  Turn on automatic check pointing at interval stored on server
                                  or with interval specified as the second argument
                        ALWAYS:   Check point at any change in node tree, *NOT* recommended for
large definitions
                        UNDEFINED:The default, which allows for immediate check pointing, or alarm
setting
  [(int)interval=120] : This specifies the interval in seconds when server should automatically
check pt.
                        This will only take effect if mode is on_time/CHECK_ON_TIME
                        Should ideally be a value greater than 60 seconds, default is 120 seconds
  [(int)alarm=30]     : Specifies check pt save alarm time. If saving the check pt takes longer
than
                        the alarm time, then the late flag is set on the server.
                        This flag will need to be cleared manually.
)
```

Note

When the time taken to save the check pt is excessive, it can interfere with job scheduling. It may be an indication of the following:

- slow disk
- file system full
- The definition is very large and needs to split

Usage:

```
try:
    ci = Client()                      # use default host(ECF_HOST) &amp;amp; port(ECF_PORT)
    ci.checkpt()                       # Save the definition held in the server to disk
    ci.checkpt(CheckPt.NEVER)          # Switch off check pointing
    ci.checkpt(CheckPt.ON_TIME)        # Start automatic check pointing at the interval stored in
the server
    ci.checkpt(CheckPt.ON_TIME,180)    # Start automatic check pointing every 180 seconds
    ci.checkpt(CheckPt.ALWAYS)         # Check point at any state change in node tree. *not*
recommended for large defs
    ci.checkpt(CheckPt.UNDEFINED,0,35) # Change check point save time alarm to 35 seconds
                                       # With these arguments mode and interval remain unchanged
except RuntimeError, e:
    print(str(e))
```

child_abort((Client)arg1[, (str)reason='']) None :

Child command,notify server job has aborted, can provide an optional reason

child_complete((Client)arg1) None :

Child command,notify server job has complete

child_event((Client)arg1, (str)arg2) None :

Child command,notify server event occurred, requires the event name

child_init((Client)arg1) None :

Child command,notify server job has started

child_label((Client)arg1, (str)arg2, (str)arg3) None :

Child command,notify server label changed, requires label name, and new value

child_meter((Client)arg1, (str)arg2, (int)arg3) None :

Child command,notify server meter changed, requires meter name and value

child_wait((Client)arg1, (str)arg2) None :

Child command,wait for expression to come true

clear_log((Client)arg1) int :

Request the ecflow_server to clear log file. Log file will be empty after this call.

Usage:

```
try:
    ci = Client()    # use default host(ECF_HOST) &amp;amp; port(ECF_PORT)
    ci.clear_log()   # log file is now empty
except RuntimeError, e:
    print(str(e))
```

debug_server_off((Client)arg1) int :

Disable server debug

debug_server_on((Client)arg1) int :

Enable server debug, Will dump to standard out on server host.

delete((Client)arg1, (str)abs_node_path[, (bool)force=False]) int :

Delete the node (s) specified.

If a node is submitted or active, then a Exception will be raised. To force the deletion at the expense of zombie creation, then set the force parameter to true

```
void delete(
   list paths          : List of paths.
   [(bool)force=False] : If true delete even if in "active" or "submitted" states
                         Which risks creating zombies.
)
void delete(
   string absolute_node_path: Path name of node to delete.
   [(bool)force=False]       : If true delete even if in "active" or "submitted" states
)
```

Usage:

```
try:
    ci = Client()                     # use default host(ECF_HOST) &amp;amp; port(ECF_PORT)
    ci.delete("/s1/f1/task1")

    paths = ["/s1/f1/t1","/s2/f1/t2"]
    ci.delete(paths)                  # delete all tasks specified in the paths
except RuntimeError, e:
    print(str(e))
```

delete((Client)arg1, (list)paths [, (bool)force=False]) -> None

delete_all((Client)arg1[, (bool)force=False]) int :

Delete all the node s held in the ecflow_server.

The suite definition in the server will be empty, after this call. **Use with care** If a node is submitted or active, then a Exception will be raised. To force the deletion at the expense of zombie creation, then set the force parameter to true

```
void delete_all(
   [(bool)force=False] : If true delete even if in "active" or "submitted" states
                         Which risks creating zombies.
)
```

Usage:

```
try:
    ci = Client()    # use default host(ECF_HOST) &amp;amp; port(ECF_PORT)
    ci.delete_all()
    ci.get_server_defs()
except RuntimeError, e:
    print(str(e));    # expect failure since all nodes deleted
```

flush_log((Client)arg1) int :

Request the ecflow_server to flush and then close log file

It is best that the server is shutdown first, as log file will be reopened whenever a command wishes to log any changes.

Usage:

```
try:
    ci = Client()    # use default host(ECF_HOST) &amp;amp; port(ECF_PORT)
    ci.flush_log()   # Log can now opened by external program
except RuntimeError, e:
    print(str(e))
```

force_event((Client)arg1, (str)arg2, (str)arg3) None :

Set or clear a event

```
void force_event(
   string absolute_node_path:event: Path name to node: &lt; event name | number&gt;
                                    The paths must begin with a leading "/"
   string signal                  : [ set | clear ]
)
void force_event(
   list paths    : A list of absolute node paths. Each path must include a event name
                   The paths must begin with a leading "/"
   string signal : [ set | clear ]
)
```

Usage:

```
try:
    ci = Client()    # use default host(ECF_HOST) &amp; port(ECF_PORT)
    ci.force_event("/s1/f1:event_name","set")

    # Set or clear a event for a list of events
    paths = [ "/s1/t1:ev1", "/s2/t2:ev2" ]
    ci.force_event(paths,"clear")
except RuntimeError, e:
    print(str(e))
```

force_event((Client)arg1, (list)arg2, (str)arg3) -> None

force_state((Client)arg1, (str)arg2, (State)arg3) None :

Force a node(s) to a given state

When a task is set to complete, it may be automatically re-queued if it has multiple time dependencies. In the specific case where a task has a single time dependency and we want to interactively set it to complete a flag is set so that it is not automatically re-queued when set to complete. The flag is applied up the node hierarchy until reach a node with a repeat or cron attribute. This behaviour allow repeat values to be incremented interactively. A repeat attribute is incremented when all the child nodes are complete in this case the child nodes are automatically re-queued

```
void force_state(
   string absolute_node_path: Path name to node. The path must begin with a leading "/"
   State::State state       : [ unknown | complete | queued | submitted | active | aborted ]
)
void force_state(
   list paths         : A list of absolute node paths. The paths must begin with a leading "/"
   State::State state : [ unknown | complete | queued | submitted | active | aborted ]
)
```

Usage:

```
try:
    ci = Client()    # use default host(ECF_HOST) &amp; port(ECF_PORT)
    # force a single node to complete
    ci.force_state("/s1/f1",State.complete)

    # force a list of nodes to complete
    paths = [ "/s1/t1", "/s1/t2", "/s1/f1/t1" ]
    ci.force_state(paths,State.complete)
except RuntimeError, e:
    print(str(e))
```

Effect:

Lets see the effect of forcing complete on the following defs

```
suite s1
   task t1; time 10:00             # will complete straight away
   task t2; time 10:00 13:00 01:00 # will re-queue 3 times and complete on fourth
```

In the last case (task t2) after each force complete, the next time slot is incremented. This can be seen by calling the Why command.

force_state((Client)arg1, (list)arg2, (State)arg3) -> None

force_state_recursive((Client)arg1, (str)arg2, (State)arg3) None :

Force node(s) to a given state recursively

```
void force_state_recursive(
   string absolute_node_path: Path name to node.The paths must begin with a leading "/"
   State::State state       : [ unknown | complete | queued | submitted | active | aborted ]
)
void force_state_recursive(
   list  paths         : A list of absolute node paths.The paths must begin with a leading "
/"
   State::State state  : [ unknown | complete | queued | submitted | active | aborted ]
)
```

Usage:

```
try:
    ci = Client()    # use default host(ECF_HOST) &amp;amp; port(ECF_PORT)
    ci.force_state_recursive("/s1/f1",State.complete)

    # recursively force a list of nodes to complete
    paths = [ "/s1", "/s2", "/s1/f1/t1" ]
    ci.force_state_recursive(paths,State.complete)
except RuntimeError, e:
    print(str(e))
```

force_state_recursive((Client)arg1, (list)arg2, (State)arg3) -> None

free_all_dep((Client)arg1, (str)arg2) None :

Free all trigger, date and all time(day, today, cron,etc) dependencies

```
void free_all_dep(
   string absolute_node_path : Path name to node
)
```

After freeing the time related dependencies (i.e.. time,today,cron) the next time slot will be missed.

Usage:

```
try:
    ci = Client()   # use default host(ECF_HOST) &amp;amp; port(ECF_PORT)
    ci.free_all_dep("/s1/task")
except RuntimeError, e:
    print(str(e))
```

free_all_dep((Client)arg1, (list)arg2) -> None

free_date_dep((Client)arg1, (str)arg2) None :

Free date dependencies for a node

```
void free_date_dep(
   string absolute_node_path : Path name to node
)
```

Usage:

```
try:
    ci = Client()   # use default host(ECF_HOST) &amp;amp; port(ECF_PORT)
    ci.free_date_dep("/s1/task")
except RuntimeError, e:
    print(str(e))
```

free_date_dep((Client)arg1, (list)arg2) -> None

free_time_dep((Client)arg1, (str)arg2) None :

Free all time dependencies. i.e.. time, day, today, cron

```
void free_time_dep(
   string absolute_node_path : Path name to node
)
```

After freeing the time related dependencies (i.e. time,today,cron) the next time slot will be missed.

Usage:

```
try:
    ci = Client()   # use default host(ECF_HOST) &amp;amp; port(ECF_PORT)
    ci.free_time_dep("/s1/task")
except RuntimeError, e:
    print(str(e))
```

free_time_dep((Client)arg1, (list)arg2) -> None

free_trigger_dep((Client)arg1, (str)arg2) None :

Free trigger dependencies for a node

```
void free_trigger_dep(
   string absolute_node_path : Path name to node
)
```

Usage:

```
try:
    ci = Client()         # use default host(ECF_HOST) &amp;amp; port(ECF_PORT)
    ci.free_trigger_dep("/s1/f1/task")
except RuntimeError, e:
    print(str(e))
```

free_trigger_dep((Client)arg1, (list)arg2) -> None

get_defs((Client)arg1) Defs :

Returns the suite definition stored on the Client.

Use `ecflow.Client.sync_local()` to retrieve the definition from the server first. The definition is *retained* in memory until the next call to sync_local().

Usage:

```
try:
    ci = Client()         # use default host(ECF_HOST) &amp; port(ECF_PORT)
    ci.sync_local()       # get the definition from the server and store on "ci"
    print(ci.get_defs())  # print out definition stored in the client
    print(ci.get_defs())  # print again, this shows that defs is retained on ci
except RuntimeError, e:
    print(str(e))
```

get_file((Client)arg1, (str)arg2, (str)arg3, (str)arg4) str :

File command can be used to request the various file types associated with a node

This command defaults to returning a max of 10000 lines. This can be changed

```
string get_file(
   string absolute_node_path    : Path name to node
   [(string)file_type="script"] : file_type = [ script&lt;default&gt; | job | jobout | manual |
kill | stat ]
   [(string)max_lines="10000"] : The number of lines in the file to return
)
```

Usage:

```
try:
    ci = Client()        # use default host(ECF_HOST) &amp; port(ECF_PORT)
    for file in [ "script", "job", "jobout", "manual", "kill", "stat" ]:
       print(ci.get_file("/suite/f1/t1",file))  # print the contents of the file
except RuntimeError, e:
   print(str(e))
```

get_file((Client)arg1, (str)arg2, (str)arg3) -> str

get_host((Client)arg1) str :

Return the host, assume set_host_port() has been set, otherwise return localhost

get_port((Client)arg1) str :

Return the port, assume set_host_port() has been set. otherwise returns 3141

get_server_defs((Client)arg1) int :

Get all suite Node tree's from the ecflow_server.

The definition is *retained* in memory until the next call to get_server_defs(). This is important since get_server_defs() could return several megabytes of data. Hence we only want to call it once, and then access it locally with get_defs(). If you need to access the server definition in a loop use `ecflow.Client.sync_local` instead since this is capable of returning incremental changes, and thus considerably reducing the network load.

Usage:

```
try:
    ci = Client()         # use default host(ECF_HOST) &amp; port(ECF_PORT)
    ci.get_server_defs()  # get the definition from the server and store on "ci"
    print(ci.get_defs())  # print out definition stored in the client
    print(ci.get_defs())  # print again, this shows that defs is retained on ci
except RuntimeError, e:
    print(str(e))
```

group((Client)arg1, (str)arg2) int :

Allows a series of commands to be executed in the ecflow_server

```
void group(
    string cmds : a list of ";" separated commands
)
```

Usage:

```
try:
    ci = Client()               # use default host(ECF_HOST) &amp; port(ECF_PORT)
    ci.group("get; show")
    ci.group("get; show state") # show node states and trigger abstract syntax trees
except RuntimeError, e:
    print(str(e))
```

halt_server((Client)arg1) int :

Halt the ecflow_server

Stop server communication with jobs, and new job scheduling, and stops check pointing. See server states

Usage:

```
try:
    ci = Client()            # use default host(ECF_HOST) &amp; port(ECF_PORT)
    ci.halt_server()
except RuntimeError, e:
    print(str(e))
```

in_sync((Client)arg1) bool :

Returns true if the definition on the client is in sync with the ecflow_server

Warning

Calling in_sync() is **only** valid after a call to sync_local().

Usage:

```
try:
   ci = Client()                     # use default host(ECF_HOST) &amp; port(ECF_PORT)
   ci.sync_local()                   # very first call gets the full Defs
   client_defs = ci.get_defs()       # End user access to the returned Defs
   ... after a period of time
   ci.sync_local()                   # Subsequent calls to sync_local() users the local Defs to
sync incrementally
   if ci.in_sync():                  # returns true  changed and changes applied to client
      print("Client is now in sync with server")
   client_defs = ci.get_defs()       # End user access to the returned Defs
except RuntimeError, e:
    print(str(e))
```

job_generation((Client)arg1, (str)arg2) int :

Job submission for chosen Node *based* on dependencies The ecflow_server traverses the node tree every 60 seconds, and if the dependencies are free does *job creation* and submission. Sometimes the user may free time/date dependencies to avoid waiting for the server poll, this commands allows early job generation

```
void job_generation(
   string absolute_node_path: Path name for job generation to start from
)
If empty string specified generates for full definition.
```

Usage:

```
try:
    ci = Client()    # use default host(ECF_HOST) &amp; port(ECF_PORT)
    ci.job_generation("/s1")  # generate jobs for suite "/s1
except RuntimeError, e:
    print(str(e))
```

kill((Client)arg1, (str)arg2) None :

Kills the job associated with the node

```
void kill(
   list paths: List of paths. Paths must begin with a leading "/" character
)
void kill(
   string absolute_node_path: Path name to node to kill.
)
```

If a family or suite is selected, will kill hierarchically. Kill uses the ECF_KILL_CMD variable. After variable substitution it is invoked as a command. The ECF_KILL_CMD variable should be written in such a way that the output is written to % ECF_JOB%.kill, i.e..:

```
kill -15 %ECF_RID% &gt; %ECF_JOB%.kill 2&gt;&amp;1
/home/ma/emos/bin/ecfkill %USER% %HOST% %ECF_RID% %ECF_JOB% &gt; %ECF_JOB%.kill 2&gt;&amp;1
```

Exceptions can be raised because:

- The absolute_node_path does not exist in the server
- ECF_KILL_CMD variable is not defined
- variable substitution fails

Usage:

```
try:
    ci = Client()    # use default host(ECF_HOST) &amp; port(ECF_PORT)
    ci.kill("/s1/f1")
    time.sleep(2)
    print(ci.file("/s1/t1","kill")) # request kill output
except RuntimeError, e:
    print(str(e))
```

kill((Client)arg1, (list)arg2) -> None

load((Client)arg1, (str)path_to_defs[, (bool)force=False[, (bool)check_only=False[, (bool) print=False]]]) int :

Load a suite definition or checkpoint file given by the file_path argument into the ecflow_server

```
void load(
   string file_path     : path name to the definition file
   [(bool)force=False]  : If true overwrite suite of same name
   [(bool)print=False]  : print parsed defs to standard out
)
```

By default throws a RuntimeError exception for errors. If force is not used and suite of the same name already exists in the server, then a error is thrown

Usage:

```
defs_file = "Hello.def"
defs = Defs()
suite = def.add_suite("s1")
family = suite.add_family("f1")
for i in [ "_1", "_2", "_3" ]:
   family.add_task( "t" + i )
defs.save_as_defs(defs_file)  # write out in memory defs into the file "Hello.def"
...
try:
    ci = Client()       # use default host(ECF_HOST) &amp;amp; port(ECF_PORT)
    ci.load(defs_file)  # open and parse defs or checkpoint file, and load into server.
except RuntimeError, e:
    print(str(e))
```

load((Client)arg1, (Defs)defs [, (bool)force=False]) -> int :

Load a in memory suite definition into the ecflow_server

```
void load(
   Defs defs           : A in memory definition
   [(bool)force=False] : for true overwrite suite of same name
)
```

If force is not used and suite already exists in the server, then a error is thrown.

Usage:

```
defs = Defs()
suite = defs.add_suite("s1")
family = suite.add_family("f1")
for i in [ "_1", "_2", "_3" ]:
    family.add_task( Task( "t" + i) )
...
try:
    ci = Client()    # use default host(ECF_HOST) &amp;amp; port(ECF_PORT)
    ci.load(defs)    # Load in memory defs, into the server
except RuntimeError, e:
    print(str(e))
```

log_msg((Client)arg1, (str)arg2) int :

Request the ecflow_server writes a string message to the log file.

Usage:

```
try:
    ci = Client()             # use default host(ECF_HOST) &amp;amp; port(ECF_PORT)
    ci.log_msg("A message") # Write message to log file
except RuntimeError, e:
    print(str(e))
```

new_log((Client)arg1[, (str)path='']) int :

Request the ecflow_server to use the path provided, as the new log file

The old log file is released.

Usage:

```
try:
    ci = Client()               # use default host(ECF_HOST) &amp;amp; port(ECF_PORT)
    ci.new_log("/path/log.log") # use "/path/log,log" as the new log file
                                # To keep track of log file Can change ECF_LOG
    ci.alter("","change","variable","ECF_LOG","/new/path.log")
    ci.new_log()
except RuntimeError, e:
    print(str(e))
```

news_local((Client)arg1) bool :

Query the ecflow_server to detect any changes.

This returns a simple bool, if there has been changes, the user should call `ecflow.Client.sync_local`. This will bring the client in sync with changes in the server. If sync_local() is not called then calling news_local() will always return true. news_local() uses the definition stored on the client:

```
bool news_local()
```

Usage:

```
try:
    ci = Client()                  # use default host(ECF_HOST) &amp;amp; port(ECF_PORT)
    if ci.news_local():            # has the server changed
       print("Server Changed")     # server changed bring client in sync with server
       ci.sync_local()             # get the full definition from the server if first time
                                   # otherwise apply incremental changes to Client definition,
                                   # bringing it in sync with the server definition
       print(ci.get_defs())        # print the synchronised definition. Should be same as server
except RuntimeError, e:
    print(str(e))
```

order((Client)arg1, (str)arg2, (str)arg3) None :

Re-orders the node s in the suite definition held by the ecflow_server

It should be noted that in the absence of dependencies, the order in which task s are submitted, depends on the order in the definition. This changes the order and hence affects the submission order

```
void order(
   string absolute_node_path: Path name to node.
   string order_type        : Must be one of [ top | bottom | alpha | order | up | down ]
)
o top     raises the node within its parent, so that it is first
o bottom  lowers the node within its parent, so that it is last
o alpha   Arranges for all the peers of selected note to be sorted alphabetically
o order   Arranges for all the peers of selected note to be sorted in reverse alphabet
o up      Moves the selected node up one place amongst its peers
o down    Moves the selected node down one place amongst its peers
```

Exceptions can be raised because:

- The absolute_node_path does not exist in the server
- The order_type is not the right type

Usage:

```
try:
    ci = Client()   # use default host(ECF_HOST) &amp;amp; port(ECF_PORT)
    ci.order("/s1/f1","top")
except RuntimeError, e:
    print(str(e))
```

ping((Client)arg1) int :

Checks if the ecflow_server is running

```
void ping()
```

The default behaviour is to check on host ‘localhost’ and port 3141 It should be noted that any Client function will fail if the server is is not running. Hence ping() is not strictly required. However its main distinction from other Client function is that it is quite fast.

Usage:

```
try:
    ci = Client("localhost","3150")
    ci.ping()
    print("------- Server already running------")
    do_something_with_server(ci)
except RuntimeError, e:
    print("------- Server *NOT* running------" + str(e))
```

plug((Client)arg1, (str)arg2, (str)arg3) int :

Plug command is used to move node s

The destination node can be on another ecflow_server. In which case the destination path should be of the form ‘//<host>:<port>/suite /family/task

```
void plug(
   string source_absolute_node_path       : Path name to source node
   string destination_absolute_node_path  : Path name to destination node. Note if only
                                            "//host:port" is specified the whole suite can be moved
)
```

By default throws a RuntimeError exception for errors.

Exceptions can be raised because:

- Source node is in a active or submitted state.
- Another user already has an lock.
- source/destination paths do not exist on the corresponding servers
- If the destination node path is empty, i.e... only host:port is specified, then the source node must correspond to a suite.
- If the source node is added as a child, then its name must be unique

Usage:

```
try:
    ci = Client()    # use default host(ECF_HOST) &amp;amp; port(ECF_PORT)
    ci.plug("/suite","host3:3141")
except RuntimeError, e:
    print(str(e))
```

reload_wl_file((Client)arg1) int :

Request that the ecflow_server reload the white list file.

The white list file if present, can be used to control who has read/write access to the ecflow_server:

```
void reload_wl_file()
```

Usage:

```
try:
    ci = Client()            # use default host(ECF_HOST) &amp;amp; port(ECF_PORT)
    ci.reload_wl_file()
except RuntimeError, e:
    print(str(e))
```

replace((Client)arg1, (str)arg2, (str)arg3, (bool)arg4, (bool)arg5) int :

Replaces a node in a suite definition with the given path. The definition is in the ecflow_server

```
void replace(
   string absolute_node_path: Path name to node in the client defs.
                              This is also the node we want to replace in the server.
   string client_defs_file  : File path to defs files, that provides the definition of the
new node
   [(bool)parent=False]     : create parent families or suite as needed,
                              when absolute_node_path does not exist in the server
   [(bool)force=False]      : check for zombies, if force = true, bypass checks
)

void replace(
   string absolute_node_path: Path name to node in the client defs.
                              This is also the node we want to replace in the server.
   Defs client_defs         : In memory client definition that provides the definition of
the new node
   [(bool)parent=False]     : create parent families or suite as needed,
                              when absolute_node_path does not exist in the server
   [(bool)force=False]      : check for zombies, force = true, bypass checks
)
```

Exceptions can be raised because:

- The absolute_node_path does not exist in the provided definition
- The provided client definition must be free of errors
- If the third argument is not provided, then the absolute_node_path must exist in the server defs
- replace will fail, if child task nodes are in active / submitted state

After replace is done, we check trigger expressions. These are reported to standard output. It is up to the user to correct invalid trigger expressions, otherwise the tasks will *not* run. Please note, you can use check() to check trigger expression and limits in the server.

Usage:

```
try:
    ci = Client()    # use default host(ECF_HOST) &amp;amp; port(ECF_PORT)
    ci.replace("/s1/f1","/tmp/defs.def")
except RuntimeError, e:
    print(str(e))

try:
    ci.replace("/s1",client_defs) # replace suite "s1" in the server, with "s1" in the
client_defs
except RuntimeError, e:
    print(str(e))
```

replace((Client)arg1, (str)arg2, (Defs)arg3, (bool)arg4, (bool)arg5) -> int

replace((Client)arg1, (str)arg2, (Defs)arg3) -> None

replace((Client)arg1, (str)arg2, (str)arg3) -> None

requeue((Client)arg1, (str)abs_node_path[, (str)option='']) None :

Re queues the specified node (s)

```
void requeue(
   list paths     : A list of paths. Node paths must begin with a leading "/" character
   [(str)option=""] : option = ("" | "abort" | "force")
       ""    : empty string, the default, re-queue the node
       abort: means re-queue only aborted tasks below node
       force: means re-queueing even if there are nodes that are active or submitted
)
void requeue(
   string absolute_node_path : Path name to node
   [(string)option=""]       : option = ("" | "abort" | "force")
)
```

Usage:

```
try:
    ci = Client()                   # use default host(ECF_HOST) &amp;amp; port(ECF_PORT)
    ci.requeue("/s1","abort")       # re-queue aborted tasks below suite /s1

    path_list = ["/s1/f1/t1","/s2/f1/t2"]
    ci.requeue(path_list)
except RuntimeError, e:
    print(str(e))
```

requeue((Client)arg1, (list)paths [, (str)option='']) -> None

reset((Client)arg1) None :

reset client definition, and handle number

restart_server((Client)arg1) int :

Restart the ecflow_server

Start job scheduling, communication with jobs, and respond to all requests. See server states

Usage:

```
try:
    ci = Client()            # use default host(ECF_HOST) &amp;amp; port(ECF_PORT)
    ci.retstart_server()
except RuntimeError, e:
    print(str(e))
```

restore_from_checkpt((Client)arg1) int :

Request the ecflow_server loads the check point file from disk

The server will first try to open file at ECF_HOME/ECF_CHECK if that fails it will then try path ECF_HOME/ECF_CHECKOLD. An error is returned if the server has not been halted or contains a suite definition

Usage:

```
try:
    ci = Client()             # use default host(ECF_HOST) &amp;amp; port(ECF_PORT)
    ci.halt_server()          # server must be halted, otherwise restore_from_checkpt will throw
    ci.restore_from_checkpt() # restore the definition from the check point file
except RuntimeError, e:
    print(str(e))
```

resume((Client)arg1, (str)arg2) None :

Resume *job creation* / generation for the given node

```
void resume(
   list paths: List of paths. Paths must begin with a leading "/" character
)
void resume(
   string absolute_node_path: Path name to node to resume.
)
```

Usage:

```
try:
    ci = Client()   # use default host(ECF_HOST) &amp; port(ECF_PORT)
    ci.resume("/s1/f1/task1")
    paths = ["/s1/f1/t1","/s2/f1/t2"]
    ci.resume(paths)
except RuntimeError, e:
    print(str(e))
```

resume((Client)arg1, (list)arg2) -> None

run((Client)arg1, (str)arg2, (bool)arg3) None :

Immediately run the jobs associated with the input node.

Ignore trigger s, limit s, suspended, time or date dependencies, just run the task. When a job completes, it may be automatically re-queued if it has multiple time dependencies. In the specific case where a task has a SINGLE time dependency and we want to avoid re running the task then a flag is set so that it is not automatically re-queued when set to complete. The flag is applied up the node hierarchy until we reach a node with a repeat or cron attribute. This behaviour allow repeat values to be incremented interactively. A repe at attribute is incremented when all the child nodes are complete in this case the child nodes are automatically re-queued

```
void run(
   string absolute_node_path : Path name to node. If the path is suite/family will recursively
                               run all child tasks
   [(bool)force=False]       : If true, run even if there are nodes that are active or submitted.
)
void run(
   list  paths               : List of paths. If the path is suite/family will recursively run all
child tasks
   [(bool)force=False]       : If true, run even if there are nodes that are active or submitted.
)
```

Usage:

```
try:
    ci = Client()                          # use default host(ECF_HOST) &amp; port(ECF_PORT)
    ci.run("/s1")                          # run all tasks under suite /s1

    ci.run(["/s1/f1/t1","/s2/f1/t2"])      # run all tasks specified
except RuntimeError, e:
    print(str(e))
```

Effect:

```
Lets see the effect of run command on the following defs::

suite s1
   task t1; time 10:00             # will complete straight away
   task t2; time 10:00 13:00 01:00 # will re-queue 3 times and complete on fourth run
```

In the last case (task t2) after each run the next time slot is incremented. This can be seen by calling the Why command.

run((Client)arg1, (list)arg2, (bool)arg3) -> None

server_version((Client)arg1) str :

Returns the server version, can throw for old servers, that did not implement this request.

set_child_password((Client)arg1, (str)arg2) None :

Set the password, needed for authentication, provided by the server using %ECF_PASS% By default the environment variable ECF_PASS is read for the jobs password This can be overridden for the python child api

set_child_path((Client)arg1, (str)arg2) None :

Set the path to the task, obtained from server using %ECF_NAME% By default the environment variable ECF_NAME is read for the task path This can be overridden for the python child api

set_child_pid((Client)arg1, (str)arg2) None :

Set the process id of this job
    By default the environment variable ECF_RID is read for the jobs process or remote id This can be overridden for the python child api
set_child_pid((Client)arg1, (int)arg2) -> None :
    Set the process id of this job By default the environment variable ECF_RID is read for the jobs process or remote id This can be overridden for the python child api

set_child_timeout((Client)arg1, (int)arg2) None :

Set timeout if child can not connect to server, default is 24 hours. The input is required to be in seconds By default the environment variable ECF_TIMEOUT is read to control how long child command should attempt to connect to the server This can be overridden for the python child api

set_child_try_no((Client)arg1, (int)arg2) None :

Set the try no, i.e.. the number of times this job has run, obtained from the server, using %ECF_TRYNO% By default the environment variable ECF_TRYNO is read to record number of times job has been run This can be overridden for the python child api

set_connection_attempts((Client)arg1, (int)arg2) None :

Set the number of times to connect to ecflow_server, in case of connection failures

The period between connection attempts is handled by Client.set_retry_connection_period(). If the network is unreliable the connection attempts can be be increased, likewise when the network is stable this number could be reduced to one. This can increase responsiveness and reduce latency. Default value is set as 2. Setting a value less than one is ignored, will default to 1 in this case:

```
set_connection_attempts(
   int attempts # must be an integer &gt;= 1
)
```

Exceptions:

- None

Usage:

```
ci = Client()
ci.set_connection_attempts(3)     # make 3 attempts for server connection
ci.set_retry_connection_period(1) # wait 1 second between each attempt
```

set_host_port((Client)arg1, (str)arg2, (str)arg3) None :

Override the default(localhost and port 3141) and environment setting(ECF_HOST and ECF_PORT)
    and set it explicitly:

```
set_host_port(
   string host, # The server name. Can not be empty.
   string port  # The port on the server, must be unique to the server
)

set_host_port(
   string host, # The server name. Can not be empty.
   int port     # The port on the server, must be unique to the server
)

set_host_port(
   string host_port, # Expect"s &lt;host&gt;:&lt;port&gt; or &lt;host&gt;@&lt;port&gt;
)
```

Exceptions:

- Raise a RuntimeError if the host or port is empty

Usage:

```
try:
    ci = Client()
    ci.set_host_port("localhost","3150")
    ci.set_host_port("avi",3150)
    ci.set_host_port("avi:3150")
except RuntimeError, e:
    print(str(e))
```

set_host_port((Client)arg1, (str)arg2) -> None

set_host_port((Client)arg1, (str)arg2, (int)arg3) -> None

set_retry_connection_period((Client)arg1, (int)arg2) None :

Set the sleep period between connection attempts

Whenever there is a connection failure we wait a number of seconds before trying again. i.e... to get round glitches in the network. For the ping command this is hard wired as 1 second. This wait between connection attempts can be configured here. i.e.. This could be reduced to increase responsiveness. Default: In debug this period is 1 second and in release mode 10 seconds:

```
set_retry_connection_period(
   int period # must be an integer &gt;= 0
)
```

Exceptions:

- None

Usage:

```
ci = Client()
ci.set_connection_attempts(3)     # make 3 attempts for server connection
ci.set_retry_connection_period(1) # wait 1 second between each attempt
```

shutdown_server((Client)arg1) int :

Shut down the ecflow_server

Stop server from scheduling new jobs. See server states

Usage:

```
try:
    ci = Client()           # use default host(ECF_HOST) &amp; port(ECF_PORT)
    ci.shutdown_server()
except RuntimeError, e:
    print(str(e))
```

sort_attributes((Client)arg1, (str)abs_node_path, (str)attribute_name[, (bool)recursive=True]) None

sort_attributes((Client)arg1, (list)paths, (str)attribute_name [, (bool)recursive=True]) -> None

stats((Client)arg1) None :

Prints the ecflow_server statistics to standard out

```
void stats()
```

Usage:

```
try:
    ci = Client()  # use default host(ECF_HOST) &amp; port(ECF_PORT)
    ci.stats()
except RuntimeError, e:
    print(str(e))
```

stats_reset((Client)arg1) None :

Resets the statistical data in the server

```
void stats_reset()
```

Usage:

```
try:
    ci = Client()  # use default host(ECF_HOST) &amp; port(ECF_PORT)
    ci.stats_reset()
except RuntimeError, e:
    print(str(e))
```

status((Client)arg1, (str)arg2) None :

Shows the status of a job associated with a task

```
void status(
   list paths: List of paths. Paths must begin with a leading "/" character
)
void status(
   string absolute_node_path
)
```

If a family or suite is selected, will invoke status command hierarchically. Status uses the ECF_STATUS_CMD variable. After variable substitution it is invoked as a command. The command should be written in such a way that the output is written to % ECF_JOB%.stat, i.e..:

```
/home/ma/emos/bin/ecfstatus  %USER% %HOST% %ECF_RID% %ECF_JOB% &gt; %ECF_JOB%.stat 2&gt;
&amp;1
```

Exceptions can be raised because:

- The absolute_node_path does not exist in the server
- ECF_STATUS_CMD variable is not defined
- variable substitution fails

Usage:

```
try:
    ci = Client()    # use default host(ECF_HOST) &amp;amp; port(ECF_PORT)
    ci.status("/s1/t1")
    time.sleep(2)
    print(ci.file("/s1/t1","stats")) # request status output
except RuntimeError, e:
    print(str(e))
```

status((Client)arg1, (list)arg2) -> None

suites((Client)arg1) list :

Returns a list strings representing the suite names

```
list(string) suites()
```

Usage:

```
try:
    ci = Client()  # use default host(ECF_HOST) &amp;amp; port(ECF_PORT)
    suites = ci.suites()
    print(suites)
except RuntimeError, e:
    print(str(e))
```

suspend((Client)arg1, (str)arg2) None :

Suspend *job creation* / generation for the given node

```
void suspend(
   list paths: List of paths. Paths must begin with a leading "/" character
)
void suspend(
   string absolute_node_path: Path name to node to suspend.
)
```

Usage:

```
try:
    ci = Client()    # use default host(ECF_HOST) &amp;amp; port(ECF_PORT)
    ci.suspend("/s1/f1/task1")
    paths = ["/s1/f1/t1","/s2/f1/t2"]
    ci.suspend(paths)
except RuntimeError, e:
    print(str(e))
```

suspend((Client)arg1, (list)arg2) -> None

sync_local((Client)arg1) int :

Requests that ecflow_server returns the full definition or incremental change made and applies them to the client Defs

When there is a very large definition, calling `ecflow.Client.get_server_defs` each time can be *very* expensive both in terms of memory, speed, and network bandwidth. The alternative is to call this function, which will get the incremental changes, and apply them local client suite definition effectively synchronising the client and server Defs. If the period of time between two sync() calls is too long, then the full server definition is returned and assigned to the client Defs. We can determine if the changes were applied by calling in_sync() after the call to sync_local():

```
void sync_local();                    # The very first call, will get the full Defs.
```

Exceptions:

- raise a RuntimeError if the delta change can not be applied.
- this could happen if the client Defs bears no resemblance to server Defs

Usage:

```
try:
    ci = Client()                     # use default host(ECF_HOST) &amp;amp; port(ECF_PORT)
    ci.sync_local()                   # Very first call gets the full Defs
    client_defs = ci.get_defs()       # End user access to the returned Defs
    ... after a period of time
    ci.sync_local()                   # Subsequent calls to sync_local() users the local Defs to
sync incrementally
    if ci.in_sync():                  # returns true server changed and changes applied to client
       print("Client is now in sync with server")
    client_defs = ci.get_defs()       # End user access to the returned Defs
except RuntimeError, e:
    print(str(e))
```

Calling sync_local() is considerably faster than calling get_server_defs() for large Definitions

terminate_server((Client)arg1) int :

Terminate the ecflow_server

Usage:

```
try:
    ci = Client()            # use default host(ECF_HOST) &amp;amp; port(ECF_PORT)
    ci.terminate_server()
except RuntimeError, e:
    print(str(e))
```

version((Client)arg1) str :

Returns the current client version

wait_for_server_reply((Client)arg1[, (int)time_out=60]) bool :

Wait for a response from the ecflow_server:

```
void wait_for_server_reply(
   int time_out     : (default = 60)
)
```

This is used to check if server has started. Typically for tests. Returns true if server(ping) replies before time out, otherwise false

Usage:

```
ci = Client()   # use default host(ECF_HOST) &amp;amp; port(ECF_PORT)
if ci.wait_for_server_reply(30):
   print("Server is alive")
else:
   print("Timed out after 30 second wait for server response.?")
```

zombie_adopt((Client)arg1, (str)arg2) int

zombie_block((Client)arg1, (str)arg2) int

zombie_fail((Client)arg1, (str)arg2) int

zombie_fob((Client)arg1, (str)arg2) int

zombie_kill((Client)arg1, (str)arg2) int

zombie_remove((Client)arg1, (str)arg2) int

```
changed_node_paths
After a call to sync_local() we can access the list of nodes that changed
The returned list consists of node paths. IF the list is empty assume that
whole definition changed. This should be expected after the first call to sync_local()
since that always retrieves the full definition from the server:

py
Midnight
void changed_node_paths()


Usage:

py
Midnight
try:
    ci = Client()                          # use default host(ECF_HOST) &amp; port(ECF_PORT)
    if ci.news_local():                    # has the server changed
       print("Server Changed")             # server changed bring client in sync with server
       ci.sync_local()                     # get the full definition from the server if first time
                                           # otherwise apply incremental changes to Client definition,
                                           # bringing it in sync with the server definition
       defs = ci.get_defs()                # get the updated/synchronised definition
       for path in ci.changed_node_paths:
           if path == "/":                 # path "/" represent change to server node/defs
              print("defs changed")        # defs state change or user variables changed
           else:
              node = defs.find_abs_node(path)

     # if changed_node_paths is empty, then assume entire definition changed
     print(defs)                          # print the synchronised definition. Should be same as server
except RuntimeError, e:
    print(str(e))
```

class ecflow.Clock

Bases: `Boost.Python.instance`

Specifies the clock type used by the suite.

Only suites can have a clock. A gain can be specified to offset from the given date.

Constructor:

```
Clock(day,month,year,hybrid)
   int day              : Specifies the day of the month  1-31
   int month            : Specifies the month 1-12
   int year             : Specifies the year &gt; 1400
   bool hybrid&lt;optional&gt;: Default = False, true means hybrid, false means real
                          by default the clock is not real

   Time will be set to midnight, use set_gain() to alter

Clock(hybrid)
   bool hybrid: true means hybrid, false means real
                by default the clock is real
   Time will be set real time of the computer
```

Exceptions:

- raises IndexError when an invalid Clock is specified

Usage:

```
suite = Suite("s1")
clock = Clock(1,1,2012,False)
clock.set_gain(1,10,True)
suite.add_clock(clock)
```

day((Clock)arg1) int :

Returns the day as an integer, range 1-31

gain((Clock)arg1) int :

Returns the gain as an long. This represents seconds

month((Clock)arg1) int :

Returns the month as an integer, range 1-12

positive_gain((Clock)arg1) bool :

Returns a boolean, where true means that the gain is positive

set_gain((Clock)arg1, (int)arg2, (int)arg3, (bool)arg4) None :

Set the gain in hours and minutes

set_gain_in_seconds((Clock)arg1, (int)arg2, (bool)arg3) None :

Set the gain in seconds

set_virtual((Clock)arg1, (bool)arg2) None :

Sets/unsets the clock as being virtual

virtual((Clock)arg1) bool :

Returns a boolean, where true means that clock is virtual

year((Clock)arg1) int :

Returns the year as an integer, > 1400

class ecflow.Complete

Bases: `Boost.Python.instance`

Add a trigger or complete expression.

This defines a dependency for a node. There can only be one trigger or complete expression dependency per node. A node with a trigger can only be activated when the trigger has expired. Triggers can reference nodes, events, meters, variables, repeats, limits and late flag A trigger holds a node as long as the expression returns false.

Exception:

- Will throw RuntimeError if first expression is added as ‘AND’ or ‘OR’ expression Like wise second and subsequent expression must have ‘AND’ or ‘OR’ booleans set

Usage:

Multiple trigger will automatically be *anded* together, If *or* is required please use bool ‘False’ as the last argument i.e..

```
task1.add( Trigger("t2 == active" ),
           Trigger("t1 == complete or t4 == complete" ),
           Trigger("t5 == active",False))
```

The trigger for task1 is equivalent to

```
t2 == active and t1 == complete or t4 == complete or t5 == active
```

Since a large number of triggers are of the form *<node> == complete* there are are short cuts, these involves a use of a list

```
task1.add( Trigger( ["t2","t3"] )) #  This is same as t2 == complete and t3 == complete
```

You can also use a node

```
task1.add( Trigger( ["t2",taskx] ))
```

If the node 'taskx' has a parent, we use the full hierarchy, hence we will get a trigger of the form

```
t2 ==complete and /suite/family/taskx == complete
```

If however node taskx has not yet been added to its parent, we use a relative name in the trigger

```
t2 ==complete and taskx == complete
```

get_expression((Complete)arg1) str :

returns the complete expression as a string

class ecflow.Cron

Bases: `Boost.Python.instance`

cron defines a repeating time dependency for a node.

Crons are repeated indefinitely.

Avoid having a cron and repeat at the same level,as both provide looping functionality

Constructor:

```
Cron()
Cron(string time_series,
     days_of_week=list of ints,   # 0-6, Sunday-Saturday
     days_of_month=list of ints,  # 1-31
     months=list of ints)         # 1-12
Cron(TimeSeries time_series,
     days_of_week=list of ints,
     days_of_month=list of ints,
     months=list of ints)
```

Exceptions:

- raises IndexError || RuntimeError when an invalid cron is specified

Usage:

```
cron = Cron("+00:00 23:00 00:30", days_of_week=[0,1,2,3,4,5,6],days_of_month=[1,2,3,4,5,6], months=
[1,2,3,4,5,6])

# Here "+" means relative to begin or re-queue time
cron = Cron("+01:30",days_of_week=[0,1,2,3,4,5,6])

cron = Cron("+00:00 23:00 00:30", days_of_week=[0,1,2],days_of_month=[4,5,6], months=[1,2,3])

start = TimeSlot(0 , 0)
finish = TimeSlot(23, 0)
incr = TimeSlot(0, 30)
ts = TimeSeries(start, finish, incr, True)  # True means relative to suite start
cron = Cron(ts, days_of_week=[0,1,2,3,4,5,6],days_of_month=[1,2,3,4,5,6], months=[1,2])

cron = Cron()
cron.set_week_days([0, 1, 2, 3, 4, 5, 6])
cron.set_days_of_month([1, 2, 3, 4, 5, 6 ])
cron.set_months([1, 2, 3, 4, 5, 6])
cron.set_time_series(ts)

cron1 = Cron()
cron1.set_time_series(1, 30, True)  # same as cron +01:30

cron2 = Cron()
cron2.set_week_days([0, 1, 2, 3, 4, 5, 6])
cron2.set_time_series("00:30 01:30 00:01")

cron3 = Cron()
cron3.set_week_days([0, 1, 2, 3, 4, 5, 6])
cron3.set_time_series("+00:30")
```

set_days_of_month((Cron)arg1, (list)arg2) None :

Specifies days of the month. Expects a list of integers with integer range 1-31

set_months((Cron)arg1, (list)arg2) None :

Specifies months. Expects a list of integers, with integer range 1-12

set_time_series((Cron)arg1, (int)hour, (int)minute[, (bool)relative=False]) None :

time_series(hour(int),minute(int),relative to suite start(bool=false)), Add a time slot

set_time_series((Cron)arg1, (TimeSeries)arg2) -> None :
    Add a time series. This will never complete
set_time_series((Cron)arg1, (TimeSlot)arg2, (TimeSlot)arg3, (TimeSlot)arg4) -> None :
    Add a time series. This will never complete
set_time_series((Cron)arg1, (str)arg2) -> None :
    Add a time series. This will never complete

set_week_days((Cron)arg1, (list)arg2) None :

Specifies days of week. Expects a list of integers, with integer range 0==Sun to 6==Sat

time((Cron)arg1) TimeSeries :

return cron time as a TimeSeries

```
days_of_month
returns a integer list of days of the month
months
returns a integer list of months of the year
week_days
returns a integer list of week days
```

class ecflow.DState

Bases: `Boost.Python.enum`

A DState is like a ecflow.State, except for the addition of SUSPENDED

Suspended stops job generation, and hence is an attribute of a Node. DState can be used for setting the default state of node when it is begun or re queued. DState is used for defining defstatus. See `ecflow.Node.add_defstatus` and `ecflow.Defstatus` The default state of a node is queued.

Usage:

```
task = ecflow.Task("t1")
task.add_defstatus(ecflow.DState.complete)   task = ecflow.Task("t2")
task += Defstatus("complete")
task = Task("t3",
            Defstatus("complete")) # create in place
```

```
aborted = ecflow.DState.aborted
active = ecflow.DState.active
complete = ecflow.DState.complete
names = {'complete': ecflow.DState.complete, 'unknown': ecflow.DState.unknown, 'aborted': ecflow.DState.
aborted, 'submitted': ecflow.DState.submitted, 'suspended': ecflow.DState.suspended, 'active': ecflow.DState.
active, 'queued': ecflow.DState.queued}
queued = ecflow.DState.queued
submitted = ecflow.DState.submitted
suspended = ecflow.DState.suspended
unknown = ecflow.DState.unknown
values = {0: ecflow.DState.unknown, 1: ecflow.DState.complete, 2: ecflow.DState.queued, 3: ecflow.DState.
aborted, 4: ecflow.DState.submitted, 5: ecflow.DState.active, 6: ecflow.DState.suspended}
```

class ecflow.Date

Bases: `Boost.Python.instance`

Used to define a date dependency.

There can be multiple Date dependencies for a node. Any of the 3 attributes, i.e... day, month, year can be wild carded using a zero If a hybrid clock is defined on a suite, any node held by a date dependency will be set to complete at the beginning of the suite, without the task ever being dispatched otherwise, the suite would never complete.

Constructor:

```
Date(string)
   string : * means wild card
Date(day,month,year)
   int day   : represents the day, zero means wild card. day &gt;= 0 &amp; day &lt; 31
   int month : represents the month, zero means wild card. month &gt;= 0 &amp; month &lt; 12
   int year  : represents the year, zero means wild card. year &gt;= 0
```

Exceptions:

- raises IndexError when an invalid date is specified

Usage:

```
date = Date(11,12,2010)  # represent 11th of December 2010
date = Date(1,0,0);      # means the first day of every month of every year
t = Task("t1",
          Date("1.*.*"));  # Create Date in place.
```

day((Date)arg1) int :

Return the day. The range is 0-31, 0 means its wild-carded

month((Date)arg1) int :

Return the month. The range is 0-12, 0 means its wild-carded

year((Date)arg1) int :

Return the year, 0 means its wild-carded

class ecflow.Day

Bases: `Boost.Python.instance`

Defines a day dependency.

There can be multiple day dependencies. If a hybrid clock is defined on a suite, any node held by a day dependency will be set to complete at the beginning of the suite, without the task ever being dispatched otherwise the suite would never complete.

Constructor:

```
Day(string)
   string: "sunday","monday",etc
Day(Days)
   Days day: Is an enumerator with represent the days of the week
```

Usage:

```
day1 = Day(Days.sunday)
t = Task("t1",
        Day("tuesday"))
```

day((Day)arg1) Days :

Return the day as enumerator

class ecflow.Days

Bases: `Boost.Python.enum`

This enum is used as argument to a `ecflow.Day` class.

It represents the days of the week

Usage:

```
day1 = Day(Days.sunday)
day2 = Day(Days.monday)
t = Task("t1",
        day1,
        day2,
        Day(Days.tuesday))
```

```
friday = ecflow.Days.friday
monday = ecflow.Days.monday
names = {'monday': ecflow.Days.monday, 'tuesday': ecflow.Days.tuesday, 'friday': ecflow.Days.friday,
'wednesday': ecflow.Days.wednesday, 'thursday': ecflow.Days.thursday, 'sunday': ecflow.Days.sunday, 'saturday':
ecflow.Days.saturday}
saturday = ecflow.Days.saturday
sunday = ecflow.Days.sunday
thursday = ecflow.Days.thursday
tuesday = ecflow.Days.tuesday
values = {0: ecflow.Days.sunday, 1: ecflow.Days.monday, 2: ecflow.Days.tuesday, 3: ecflow.Days.wednesday, 4:
ecflow.Days.thursday, 5: ecflow.Days.friday, 6: ecflow.Days.saturday}
wednesday = ecflow.Days.wednesday
```

class ecflow.Defs

Bases: `Boost.Python.instance`

The Defs class holds the suite definition structure.

It contains all the `ecflow.Suite` and hence acts like the root for suite node tree hierarchy. The definition can be kept as python code, alternatively it can be saved as a flat ASCII definition file. If a definition is read in from disk, it will by default, check the trigger expressions. If however the definition is created in python, then checking should be done explicitly:

Defs(string)
string - The Defs class take one argument which represents the file name
Defs(Suite | Edit)
`ecflow.Suite`- One or more suites

`ecflow.Edit` - specifies user defined server variables

Example:

```
 # Build definition using Constructor approach, This allows indentation, to show the structure
 # This is a made up example to demonstrate suite construction:
defs = Defs(
    Edit(SLEEP=10,FRED="bill"),  # user defined server variables
    Suite("s1"
        Clock(1, 1, 2010, False),
        Autocancel(1, 10, True),
        Task("t1"
            Edit({"a":"12", "b":"bb"}, c="v",d="b"),
            Edit(g="d"),
            Edit(h=1),
            Event(1),
            Event(11,"event"),
            Meter("meter",0,10,10),
            Label("label","c"),
            Trigger("1==1"),
            Complete("1==1"),
            Limit("limit",10),Limit("limit2",10),
            InLimit("limitName","/limit",2),
            Defstatus(DState.complete),
            Today(0,30),Today("00:59"),Today("00:00 11:30 00:01"),
            Time(0,30),Time("00:59"),Time("00:00 11:30 00:01"),
            Day("sunday"),Day(Days.monday),
            Date(1,1,0),Date(28,2,1960),
            Autocancel(3)
            ),
        [ Family("f{}".format(i)) for i in range(1,6)]))

 defs.save_as_defs("filename.def")  # save defs into file

 defs = Defs()                      # create an empty defs
 suite = defs.add_suite("s1")
 family = suite.add_family("f1")
 for i in [ "_1", "_2", "_3" ]: family.add_task( "t" + i )
 defs.save_as_defs("filename.def")  # save defs into file
```

Create a Defs from an existing file on disk.

```
defs = Defs("filename.def")   #  Will open and parse the file and create the Definition
print(defs)
```

add()

object add(tuple args, dict kwds) :
add(..) provides a way to append Nodes and attributes

This is best illustrated with an example:

```
defs = Defs().add(
    Suite("s1").add(
        Clock(1, 1, 2010, False),
        Autocancel(1, 10, True),
        Task("t1").add(
            Edit({"a":"12", "b":"bb"}, c="v",d="b"),
            Edit(g="d"),
            Edit(h=1),
            Event(1),
            Event(11,"event"),
            Meter("meter",0,10,10),
            Label("label","c"),
            Trigger("1==1"),
            Complete("1==1"),
            Limit("limit",10),Limit("limit2",10),
            InLimit("limitName","/limit",2),
            Defstatus(DState.complete),
            Today(0,30),Today("00:59"),Today("00:00 11:30 00:01"),
            Time(0,30),Time("00:59"),Time("00:00 11:30 00:01"),
            Day("sunday"),Day(Days.monday),
            Date(1,1,0),Date(28,2,1960),
            Autocancel(3)
            ),
        [ Family("f{}".format(i)) for i in range(1,6)]))
```

We can also use '+=' with a list here are a few examples:

```
defs = Defs();
defs += [ Suite("s2"),Edit({ "x1":"y", "aa1":"bb"}, a="v",b="b") ]
```

```
defs += [ Suite("s{}".format(i)) for i in range(1,6) ]
```

```
defs = Defs()
defs += [ Suite("suite").add(
             Task("x"),
             Family("f").add( [ Task("t{}".format(i)) for i in range(1,6)] ),
             Task("y"),
             [ Family("f{}".format(i)) for i in range(1,6) ],
             Edit(a="b"),
             [ Task("t{}".format(i)) for i in range(1,6) ],
             )]
```

It is also possible to use '+'

```
defs = Defs() + Suite("s1")
defs.s1 += Autocancel(1, 10, True)
defs.s1 += Task("t1") + Edit({ "e":1, "f":"bb"}) +\
           Event(1) + Event(11,"event") + Meter("meter",0,10,10) + Label("label","c") +
Trigger("1==1") +\
           Complete("1==1") + Limit("limit",10) + Limit("limit2",10) + InLimit("limitName","
/limit",2) +\
           Defstatus(DState.complete) + Today(0,30) + Today("00:59") + Today("00:00 11:30
00:01") +\
           Time(0,30) + Time("00:59") + Time("00:00 11:30 00:01") + Day("sunday") + Day
(Days.monday) +\
           Date(1,1,0) + Date(28,2,1960) + Autocancel(3)
```

Warning

We can only use '+' when the left most object is a node, i.e.. Task('t1') in this case

add_extern((Defs)arg1, (str)arg2) None :

extern refer to nodes that have not yet been defined typically due to cross suite dependencies

trigger and complete expression s may refer to paths, and variables in other suites, that have not been loaded yet. The references to node paths and variable must exist, or exist as externs Externs can be added manually or automatically.

Manual Method:

```
void add_extern(string nodePath )
```

Usage:

```
defs = Defs("file.def")
....
defs.add_extern("/temp/bill:event_name")
defs.add_extern("/temp/bill:meter_name")
defs.add_extern("/temp/bill:repeat_name")
defs.add_extern("/temp/bill:edit_name")
defs.add_extern("/temp/bill")
```

Automatic Method:

This will scan all trigger and complete expressions, looking for paths and variables that have not been defined. The added benefit of this approach is that duplicates will not be added. It is the user's responsibility to check that extern's are eventually defined otherwise trigger expression will not evaluate correctly

```
void auto_add_externs(bool remove_existing_externs_first )
```

Usage:

```
defs = Defs("file.def")
...
defs.auto_add_externs(True)   # remove existing extern first.
```

add_suite((Defs)arg1, (Suite)arg2) Suite :

Add a suite node. See `ecflow.Suite`

If a new suite is added which matches the name of an existing suite, then an exception is thrown.

Exception:

- Throws RuntimeError is the suite name is not valid
- Throws RuntimeError if duplicate suite is added

Usage:

```
defs = Defs()               # create a empty defs
suite = Suite("suite")      # create a stand alone Suite
defs.add_suite(suite)       # add suite to defs
s2 = defs.add_suite("s2")   # create a suite and add to defs

# Alternatively we can create Suite in place
defs = Defs(
         Suite("s1",
            Family("f1",
               Task("t1"))),
         Suite("s2",
            Family("f1",
               Task("t1"))))
```

add_suite((Defs)arg1, (str)arg2) -> Suite :

Create a empty Defs

add_variable((Defs)arg1, (str)arg2, (str)arg3) Defs :

Adds a name value variable. Also see `ecflow.Edit`

This defines a variable for use in variable substitution in a ecf script file. There can be any number of variables. The variables are names inside a pair of '%' characters in an ecf script. The name are case sensitive. Special character in the value, must be placed inside single quotes if misinterpretation is to be avoided. The value of the variable replaces the variable name in the ecf script at *job creation* time. The variable names for any given node must be unique. If duplicates are added then the the last value added is kept.

Exception:

- Writes warning to standard output, if a duplicate variable name is added

Usage:

```
task.add_variable( Variable("ECF_HOME","/tmp/"))
task.add_variable( "TMPDIR","/tmp/")
task.add_variable( "COUNT",2)
a_dict = { "name":"value", "name2":"value2", "name3":"value3" }
task.add_variable(a_dict)
```

add_variable((Defs)arg1, (str)arg2, (int)arg3) -> Defs

add_variable((Defs)arg1, (Variable)arg2) -> Defs

add_variable((Defs)arg1, (dict)arg2) -> Defs

auto_add_externs((Defs)arg1, (bool)arg2) None :

extern refer to nodes that have not yet been defined typically due to cross suite dependencies

trigger and complete expression s may refer to paths, and variables in other suites, that have not been loaded yet. The references to node paths and variable must exist, or exist as externs Externs can be added manually or automatically.

Manual Method:

```
void add_extern(string nodePath )
```

Usage:

```
defs = Defs("file.def")
....
defs.add_extern("/temp/bill:event_name")
defs.add_extern("/temp/bill:meter_name")
defs.add_extern("/temp/bill:repeat_name")
defs.add_extern("/temp/bill:edit_name")
defs.add_extern("/temp/bill")
```

Automatic Method:

This will scan all trigger and complete expressions, looking for paths and variables that have not been defined. The added benefit of this approach is that duplicates will not be added. It is the user's responsibility to check that extern's are eventually defined otherwise trigger expression will not evaluate correctly

```
void auto_add_externs(bool remove_existing_externs_first )
```

Usage:

```
defs = Defs("file.def")
...
defs.auto_add_externs(True)   # remove existing extern first.
```

check((Defs)arg1) str :

Check trigger and complete expression s and limit s

- Client Side: The client side can specify externs. Hence all node path references in trigger expressions, and inlimit references to l imit s, that are unresolved and which do *not* appear in extern s are reported as errors
- Server Side: The server does not store externs. Hence all unresolved references are reported as errors

Returns a non empty string for any errors or warning

Usage:

```
# Client side
defs = Defs("my.def")        # Load my.def from disk
....
print(defs.check()) # do the check

# Server Side
try:
    ci = Client()             # use default host(ECF_HOST) &amp;amp; port(ECF_PORT)
    print(ci.check("/suite"))
except RuntimeError, e:
    print(str(e))
```

check_job_creation((Defs)arg1[, (bool)throw_on_error=False[, (bool)verbose=False]]) str :

Check *job creation* .

Will check the following:

- ecf script files and includes files can be located
- recursive includes
- manual and comments pre-processing
- variable substitution

Some task s are dummy tasks have no associated ecf script file. To disable error message for these tasks please add a variable called ECF_DUMMY_TASK to them. Checking is done in conjunction with the class `ecflow.JobCreationCtrl`. If no node path is set on class JobCreationCtrl then all tasks are checked. In the case where we want to check all tasks, use the convenience function that take no arguments.

Usage:

```
defs = Defs("my.def")                    # specify the defs we want to check, load into memory
...
print(defs.check_job_creation())         # Check job generation for all tasks
...

# throw on error and Output the tasks as they are being checked
defs.check_job_creation(throw_on_error=TrueTrue,verbose=True)

job_ctrl = JobCreationCtrl()
defs.set_verbose(True)                   # Output the tasks as they are being checked
defs.check_job_creation(job_ctrl)        # Check job generation for all tasks, same as above
print(job_ctrl.get_error_msg())
...
job_ctrl = JobCreationCtrl()
job_ctrl.set_node_path("/suite/to_check") # will hierarchically check job creation under this node
defs.check_job_creation(job_ctrl)        # job files generated to ECF_JOB
print(job_ctrl.get_error_msg())
...
job_ctrl = JobCreationCtrl()             # no set_node_path() hence check job creation for all
tasks
job_ctrl.set_dir_for_job_creation(tmp)   # generate jobs file under this directory
defs.check_job_creation(job_ctrl)
print(job_ctrl.get_error_msg())
...
job_ctrl = JobCreationCtrl()             # no set_node_path() hence check job creation for all
tasks
job_ctrl.generate_temp_dir()             # automatically generate directory for job file
defs.check_job_creation(job_ctrl)
print(job_ctrl.get_error_msg())
```

check_job_creation((Defs)arg1, (JobCreationCtrl)arg2) -> None

delete_variable((Defs)arg1, (str)arg2) None :

An empty string will delete all user variables

find_abs_node((Defs)arg1, (str)arg2) Node :

Given a path, find the the node

find_suite((Defs)arg1, (str)arg2) Suite :

Given a name, find the corresponding suite

generate_scripts((Defs)arg1) None :

Automatically generate template ecf script s for this definition Will automatically add child command s for event, meter and label s. This allows the definition to be refined with out worrying about the scripts. However it should be noted that, this will create a lot of *duplicated* script contents i.e.. in the absence of event s, meter s and label s, most of generated ecf script files will be the same. Hence should only be used an aid to debugging the definition. It uses the contents of the definition to parameterise what gets generated, and the location of the files. Will throw Exceptions for errors.

Requires:

- ECF_HOME: specified and accessible for all Tasks, otherwise RuntimeError is raised
- ECF_INCLUDE: specifies location for head.h and tail.h includes, will use angle brackets,
  i.e.. %include <head.h>, if the head.h and tail.h already exist they are used otherwise they are generated

Optional:

- ECF_FILES: If specified, then scripts are generated under this directory otherwise ECF_HOME is used.
  The missing directories are automatically created.
- ECF_CLIENT_EXE_PATH: if specified child command will use this, otherwise will use ecflow_client
  and assume this accessible on the path.
- ECF_DUMMY_TASK: Will not generated scripts for this task.
- SLEEP: Uses this variable to delay time between calls to child commands, if not specified uses delay of one second

Usage:

```
defs = ecflow.Defs()
suite = defs.add_suite("s1")
suite.add_variable("ECF_HOME","/user/var/home")
suite.add_variable("ECF_INCLUDE","/user/var/home/includes")
for i in range(1,7) :
   fam = suite.add_family("f" + str(i))
   for t in ( "a", "b", "c", "d", "e" ) :
     fam.add_task(t);
defs.generate_scripts()   # generate ".ecf" and head.h/tail.h if required
```

get_all_nodes((Defs)arg1) NodeVec :

Returns all the node s in the definition

get_all_tasks((Defs)arg1) TaskVec :

Returns all the task nodes

get_server_state((Defs)arg1) SState :

Returns the ecflow_server state: See server states

Usage:

```
try:
    ci = Client()           # use default host(ECF_HOST) &amp;amp; port(ECF_PORT)
    ci.shutdown_server()
    ci.sync_local()
    assert ci.get_defs().get_server_state() == SState.SHUTDOWN, "Expected server to be shutdown"
except RuntimeError, e:
    print(str(e))
```

get_state((Defs)arg1) State

has_time_dependencies((Defs)arg1) bool :

returns True if the suite definition has any time dependencies

restore_from_checkpt((Defs)arg1, (str)arg2) None :

Restore the suite definition from a check point file stored on disk

save_as_checkpt((Defs)arg1, (str)arg2) None :

Save the in memory suite definition as a check point file. This includes all node state.

save_as_defs((Defs)arg1, (str)arg2[, (Style)arg3]) None :

Save the in memory suite definition into a file. The file name must be passed as an argument

simulate((Defs)arg1) str :

Simulates a suite definition, allowing you predict/verify the behaviour of your suite in few seconds

The simulator will analyse the definition, and simulate the ecflow server. Allowing time dependencies that span several months, to be simulated in a few seconds. Ecflow allows the use of verify attributes. This example show how we can verify the number of times a task should run, given a start(optional) and end time(optional):

```
suite cron3              # use real clock otherwise clock starts when the simulations starts.
   clock real  1.1.2006  # define a start date for deterministic behaviour
   endclock   13.1.2006  # When to finish. end clock is *only* used for the simulator
   family cronFamily
      task t
         cron -d 10,11,12   10:00 11:00 01:00  # run on 10,11,12 of the month at 10am and 11am
         verify complete:6                     # task should complete 6 times between 1.1.2006 -
&gt; 13.1.2006
   endfamily
endsuite
```

Please note, for deterministic behaviour, the start and end clock should be specified. However if no ‘endclock’ is specified the simulation will assume the following defaults.

- No time dependencies: 24 hours
- time || today : 24 hours
- day : 1 week
- date : 1 month
- cron : 1 year
- repeat : 1 year

If there no time dependencies with an minute resolution, then the simulator will by default use 1 hour resolution. This needs to be taken into account when specifying the verify attribute If the simulation does not complete it creates defs.flat and defs.depth files. This provides clues as to the state of the definition at the end of the simulation

Usage:

```
defs = Defs("my.def")        # specify the defs we want to simulate
....
theResults = defs.simulate()
print(theResults)
```

sort_attributes((Defs)arg1, (str)attribute_type[, (bool)recursive=True]) None

sort_attributes((Defs)arg1, (AttrType)attribute_type [, (bool)recursive=True]) -> None

```
externs
Returns a list of extern s
server_variables
Returns a list of server variable s
suites
Returns a list of suite s
user_variables
Returns a list of user defined variable s
```

class ecflow.Defstatus

Bases: `Boost.Python.instance`

A node can be set with a default status other the queued

The default state of a node is queued. This defines the state to take at 'begin' or 're-queue' time See `ecflow.Node.add_defstatus` and `ecflow.DState`

state((Defstatus)arg1) DState

class ecflow.Ecf

Bases: `Boost.Python.instance`

Singleton used to control ecf debugging

static debug_equality() bool :

Returns true if debugging of equality is enabled

static debug_level() int :

Returns integer showing debug level. debug_level > 0 will disable some warning messages

static set_debug_equality((bool)arg1) None :

Set debugging for equality

static set_debug_level((int)arg1) None :

Set debug level. debug_level > 0 will disable some warning messages

class ecflow.Edit

Bases: `Boost.Python.instance`

Defines a variable on a node for use in ecf script.

A Node can have a number of variables. These variables can be added at any node level: Defs, suite, family or task. The variables are names inside a pair of '%' characters in an ecf script. The content of a variable replaces the variable name in the ecf script at job submission time. When a variable is needed at submission time, it is first sought in the task itself. If it is not found, it is sought from the tasks parent and so on, up through the node levels until found. See variable inheritance A undefined variable in a ecf script, causes the task to be aborted, without the job being submitted.

Constructor:

```
Variable(name,value)
   string name: the name of the variable
   string value: The value of the variable

Edit(dict,kwarg) # alternative that allows multiple variables
```

Usage:

```
...
var = Variable("ECF_JOB_CMD","/bin/sh %ECF_JOB% &amp;amp;")
task.add_variable(var)
task.add_variable("JOE","90")
```

The following use example of using Edit, which allow multiple variables to added at the same time

```
t = Task("t1",
          Edit({ "a":"y", "b":"bb"}, c="v",d="b"),
          Edit({ "e":1100, "f":"bb"}),
          Edit(g="d"),
          Edit(h="1"))
```

```
defs = Defs(
          Suite("s1"),
          Edit(SLEEP="1")) # Add user variable to definition
defs.s1 += [ Task("a") ]
defs.s1.a += [ Edit({ "x1":"y", "aa1":"bb"}, a="v",b="b"),
               Edit({ "var":10, "aa":"bb"}),
               Edit(d="d") ]
```

class ecflow.Event

Bases: `Boost.Python.instance`

event s are used as signal mechanism.

Typically they would be used to signal partial completion of a task and to be able to trigger another job, which is waiting for this partial completion. Only tasks can have events that are automatically set via a child command s, see below. Events are cleared automatically when a node is re-queued or begun. Suites and Families can have events, but these events must be set via the Alter command Multiple events can be added to a task. An Event has a number and a optional name. Events are typically used in trigger and complete expression , to control job creation. Event are fired within a script/job file, i.e...:

```
ecflow_client --init=$$
ecflow_client --event=foo
ecflow_client --complete
```

Hence the defining of an event for a task, should be followed with the addition of ecflow_client –event child command in the corresponding ecf script file.

Constructor:

```
Event(number, optional&lt;name = ""&gt;)
   int number            : The number must be &gt;= 0
   string name&lt;optional&gt; : If name is given, can only refer to Event by its name
```

Usage:

```
event = Event(2,"event_name")
task.add_event(event)
task1.add_event("2")          # create a event "1" and add to the task
task2.add_event("name")       # create a event "name" and add to task

# Events can be created in the Task constructor, like any other attribute
t = Task("t3",
         Event(2,"event_name"))
```

empty((Event)arg1) bool :

Return true if the Event is empty. Used when returning a NULL Event, from a find

name((Event)arg1) str :

Return the Events name as string. If number supplied name may be empty.

name_or_number((Event)arg1) str :

returns name or number as an string

number((Event)arg1) int :

Return events number as a integer. If not specified return max integer value

value((Event)arg1) bool :

Return events current value

class ecflow.Expression

Bases: `Boost.Python.instance`

Expression holds trigger or complete expression. Also see `ecflow.Trigger`

Expressions can contain references to events, meters, user variables,repeat variables and generated variables. Expressions hold a list of part expressions. This allows us to split a large trigger or complete expression into smaller ones.

Constructor:

```
Expression( expression )
   string expression  : This typically represents the complete expression
                        however part expression can still be added
Expression( part )
   PartExpression part: The first part expression should have no "and/or" set
```

Usage: To add simple expression this class can be by passed, i.e... can use:

```
task = Task("t1")
task.add_trigger( "t2 == active" )
task.add_complete( "t2 == complete" )

task = Task("t2")
task.add_trigger( "t1 == active" )
task.add_part_trigger( "t3 == active", True)
```

To store and add large expressions use a Expression with PartExpression:

```
big_expr = Expression( PartExpression("t1 == complete or t4 == complete") )
big_expr.add( PartExpression("t5 == active",True) )
big_expr.add( PartExpression("t7 == active",False) )
task.add_trigger( big_expr)
```

In the example above the trigger for task is equivalent to ‘t1 == complete or t4 == complete and t5 == active or t7 == active’

```
big_expr2 = Expression("t0 == complete"))
big_expr2.add( PartExpression("t1 == complete or t4 == complete",True) )
big_expr2.add( PartExpression("t5 == active",False) )
task2.add_trigger( big_expr2)
```

Here the trigger for task2 is equivalent to ‘t0 == complete and t1 == complete or t4 == complete or t5 == active’

add((Expression)arg1, (PartExpression)arg2) None :

Add a part expression, the second and subsequent part expressions must have ‘and/or’ set

get_expression((Expression)arg1) str :

returns the complete expression as a string

```
parts
Returns a list of PartExpression&#8217;s
```

class ecflow.Family

Bases: `ecflow.NodeContainer`

Create a family node.A Family node lives inside a suite or another family

A family is used to collect task s together or to group other families. Typically you place tasks that are related to each other inside the same family analogous to the way you create directories to contain related files. There are two ways of adding a family, see example below.

Constructor:

```
Family(name, Nodes | Attributes)
    string name : The Family name. name must consist of alpha numeric characters or
                  underscore or dot. The first character can not be dot, as this
                  will interfere with trigger expressions. Case is significant
    Nodes | Attributes: (optional)
```

Exception:

- Throws a RuntimeError if the name is not valid
- Throws a RuntimeError if a duplicate family is added

Usage:

```
suite = Suite("suite_1")       # create a suite
family = Family("family_1")    # create a family
suite.add_family(family)       # add created family to a suite
f2 = suite.add_family("f2")    # create a family f2 and add to suite

# create in place
defs = Defs(
         Suite("s1",
            Family("f1",
               Task("t1",
                   Edit(SLEEP="10")))))
```

class ecflow.FamilyVec

Bases: `Boost.Python.instance`

Hold a list of family nodes

append((FamilyVec)arg1, (object)arg2) None

extend((FamilyVec)arg1, (object)arg2) None

class ecflow.File

Bases: `Boost.Python.instance`

Utility class, Used in test only.

static build_dir() str :

Path name to ecflow build directory

static find_client() str :

Provides pathname to the client

static find_server() str :

Provides pathname to the server

static source_dir() str :

Path name to ecflow source directory

class ecflow.Flag

Bases: `Boost.Python.instance`

Represents additional state associated with a Node.

clear((Flag)arg1, (FlagType)arg2) None :

Clear the given flag. Used in test only

is_set((Flag)arg1, (FlagType)arg2) bool :

Queries if a given flag is set

static list() FlagTypeVec :

Returns the list of all flag types. returns FlagTypeVec. Used in test only

reset((Flag)arg1) None :

Clears all flags. Used in test only

set((Flag)arg1, (FlagType)arg2) None :

Sets the given flag. Used in test only

static type_to_string((FlagType)arg1) str :

Convert type to a string. Used in test only

class ecflow.FlagType

Bases: `Boost.Python.enum`

Flags store state associated with a node

- FORCE_ABORT - Node* do not run when try_no > ECF_TRIES, and task aborted by user
- USER_EDIT - task
- TASK_ABORTED - task*
- EDIT_FAILED - task*
- JOBCMD_FAILED - task*
- NO_SCRIPT - task*
- KILLED - task* do not run when try_no > ECF_TRIES, and task killed by user
- MIGRATED - Node
- LATE - Node attribute, Task is late, or Defs checkpt takes to long
- MESSAGE - Node
- BYRULE - Node*, set if node is set to complete by complete trigger expression
- QUEUELIMIT - Node
- WAIT - task*
- LOCKED - Server
- ZOMBIE - task*
- NO_REQUE - task
- NOT_SET

```
byrule = ecflow.FlagType.byrule
edit_failed = ecflow.FlagType.edit_failed
force_abort = ecflow.FlagType.force_abort
jobcmd_failed = ecflow.FlagType.jobcmd_failed
killed = ecflow.FlagType.killed
late = ecflow.FlagType.late
locked = ecflow.FlagType.locked
message = ecflow.FlagType.message
migrated = ecflow.FlagType.migrated
names = {'migrated': ecflow.FlagType.migrated, 'queuelimit': ecflow.FlagType.queuelimit, 'user_edit': ecflow.
FlagType.user_edit, 'zombie': ecflow.FlagType.zombie, 'edit_failed': ecflow.FlagType.edit_failed,
'task_aborted': ecflow.FlagType.task_aborted, 'late': ecflow.FlagType.late, 'force_abort': ecflow.FlagType.
force_abort, 'not_set': ecflow.FlagType.not_set, 'no_reque': ecflow.FlagType.no_reque, 'no_script': ecflow.
FlagType.no_script, 'locked': ecflow.FlagType.locked, 'message': ecflow.FlagType.message, 'byrule': ecflow.
FlagType.byrule, 'jobcmd_failed': ecflow.FlagType.jobcmd_failed, 'killed': ecflow.FlagType.killed, 'wait':
ecflow.FlagType.wait}
no_reque = ecflow.FlagType.no_reque
no_script = ecflow.FlagType.no_script
not_set = ecflow.FlagType.not_set
queuelimit = ecflow.FlagType.queuelimit
task_aborted = ecflow.FlagType.task_aborted
user_edit = ecflow.FlagType.user_edit
values = {0: ecflow.FlagType.force_abort, 1: ecflow.FlagType.user_edit, 2: ecflow.FlagType.task_aborted, 3:
ecflow.FlagType.edit_failed, 4: ecflow.FlagType.jobcmd_failed, 5: ecflow.FlagType.no_script, 6: ecflow.FlagType.
killed, 7: ecflow.FlagType.migrated, 8: ecflow.FlagType.late, 9: ecflow.FlagType.message, 10: ecflow.FlagType.
byrule, 11: ecflow.FlagType.queuelimit, 12: ecflow.FlagType.wait, 13: ecflow.FlagType.locked, 14: ecflow.
FlagType.zombie, 15: ecflow.FlagType.no_reque, 16: ecflow.FlagType.not_set}
wait = ecflow.FlagType.wait
zombie = ecflow.FlagType.zombie
```

class ecflow.FlagTypeVec

Bases: `Boost.Python.instance`

Hold a list of flag types

append((FlagTypeVec)arg1, (object)arg2) None

extend((FlagTypeVec)arg1, (object)arg2) None

class ecflow.InLimit

Bases: `Boost.Python.instance`

inlimit is used in conjunction with limit to provide simple load management:

```
suite x
   limit fast 1
   family f
      inlimit /x:fast
      task t1
      task t2
```

Here ‘fast’ is the name of `ecflow.Limit` and the number defines the maximum number of tasks that can run simultaneously using this limit. Thats why you do not need a trigger between tasks ‘t1’ and ‘t2’. There is no need to change the tasks. The jobs are created in the order they are defined

Constructor:

```
InLimit(name, optional&lt;path = ""&gt;, optional&lt;token = 1&gt;)
   string name           : The name of the referenced Limit
   string path&lt;optional&gt; : The path to the Limit, if this is left out, then Limit of "name" must
be specified
                           some where up the parent hierarchy
   int value&lt;optional&gt;   : The usage of the Limit. Each job submission will consume "value" tokens
                           from the Limit. defaults to 1 if no value specified.
```

Usage:

```
inlimit = InLimit("fast","/x/f", 2)
 ...
family = Family("f1",
                InLimit("mars","/x/f", 2)) # create InLimit in Node constructor
family.add_inlimit(inlimit)                # add existing inlimit using function
```

name((InLimit)arg1) str :

Return the inlimit name as string

path_to_node((InLimit)arg1) str :

Path to the node that holds the limit, can be empty

tokens((InLimit)arg1) int :

The number of token to consume from the Limit

class ecflow.JobCreationCtrl

Bases: `Boost.Python.instance`

The class JobCreationCtrl is used in *job creation* checking

Constructor:

```
JobCreationCtrl()
```

Usage:

```
defs = Defs("my.def")                     # specify the definition we want to check, load into memory
job_ctrl = JobCreationCtrl()
job_ctrl.set_node_path("/suite/to_check") # will hierarchically check job creation under this node
defs.check_job_creation(job_ctrl)         # job files generated to ECF_JOB
print(job_ctrl.get_error_msg())           # report any errors in job generation

job_ctrl = JobCreationCtrl()              # no set_node_path() hence check job creation for all tasks
job_ctrl.set_dir_for_job_creation(tmp)    # generate jobs file under this directory
defs.check_job_creation(job_ctrl)
print(job_ctrl.get_error_msg())

job_ctrl = JobCreationCtrl()              # no set_node_path() hence check job creation for all tasks
job_ctrl.generate_temp_dir()              # automatically generate directory for job file
defs.check_job_creation(job_ctrl)
print(job_ctrl.get_error_msg())
```

generate_temp_dir((JobCreationCtrl)arg1) None :

Automatically generated temporary directory for job creation. Directory written to stdout for information

get_dir_for_job_creation((JobCreationCtrl)arg1) str :

Returns the directory set for job creation

get_error_msg((JobCreationCtrl)arg1) str :

Returns an error message generated during checking of job creation

set_dir_for_job_creation((JobCreationCtrl)arg1, (str)arg2) None :

Specify directory, for job creation

set_node_path((JobCreationCtrl)arg1, (str)arg2) None :

The node we want to check job creation for. If no node specified check all tasks

set_verbose((JobCreationCtrl)arg1, (bool)arg2) None :

Output each task as its being checked.

class ecflow.Label

Bases: `Boost.Python.instance`

A label has a name and value and provides a way of displaying information in a GUI.

The value can be anything(ASCII) as it can not be used in triggers. The value of the label is set to be the default value given in the definition when the suite is begun. This is useful in repeated suites: A task sets the label to be something.

Labels can be set at any level: Suite,Family,Task. There are two ways of updating the label

- A child command can be used to automatically update the label on a task
- Using the alter command, the labels on suite family and task can be changed manually

Constructor:

```
Label(name,value)
   string name:  The name of the label
   string value: The value of the label
```

Usage:

```
t1 = Task("t1",
          Label("name","value"),  # create Labels in-place
          Label("a","b"))
t1.add_label("l1","value")
t1.add_label(Label("l2","value2"))
for label in t1.labels:
   print(label)
```

empty((Label)arg1) bool :

Return true if the Label is empty. Used when returning a NULL Label, from a find

name((Label)arg1) str :

Return the label name as string

new_value((Label)arg1) str :

Return the new label value as string

value((Label)arg1) str :

Return the original label value as string

class ecflow.Late

Bases: `Boost.Python.instance`

Sets the late flag.

When a Node is classified as being late, the only action ecflow_server can take is to set a flag. The GUI will display this alongside the node name as a icon. Only one Late attribute can be specified on a Node.

Constructor:

```
Late()
Late(kwargs)
```

Usage:

```
# This is interpreted as: The node can stay `submitted`_ for a maximum of 15 minutes
# and it must become `active`_ by 20:00 and the run time must not exceed 2 hours
late = Late()
late.submitted( 0,15 )
late.active(   20,0 )
late.complete(  2,0, true )

late = Late(submitted="00:15",active="20:00",complete="+02:00")
t = Task("t1",
         Late(submitted="00:15",active="20:00"))
```

active((Late)arg1, (int)arg2, (int)arg3) None :

active(hour,minute): The time the node must become active. If the node is still queued or submitted
by the time specified, the late flag is set
active((Late)arg1, (TimeSlot)arg2) -> None :
active(TimeSlot):The time the node must become active. If the node is still queued or submitted by the time specified, the late flag is set
active((Late)arg1) -> TimeSlot :
Return the active time as a TimeSlot

complete((Late)arg1, (int)arg2, (int)arg3, (bool)arg4) None :

complete(hour,minute):The time the node must become complete. If relative, time is taken from the time
the node became active, otherwise node must be complete by the time given
complete((Late)arg1, (TimeSlot)arg2, (bool)arg3) -> None :
complete(TimeSlot): The time the node must become complete. If relative, time is taken from the time the node became active, otherwise node must be complete by the time given
complete((Late)arg1) -> TimeSlot :
Return the complete time as a TimeSlot

complete_is_relative((Late)arg1) bool :

Returns a boolean where true means that complete is relative

is_late((Late)arg1) bool :

Return True if late

submitted((Late)arg1, (TimeSlot)arg2) None :

submitted(TimeSlot):The time node can stay submitted. Submitted is always relative. If the node stays
submitted longer than the time specified, the late flag is set
submitted((Late)arg1, (int)arg2, (int)arg3) -> None :
submitted(hour,minute) The time node can stay submitted. Submitted is always relative. If the node stays submitted longer than the time specified, the late flag is set
submitted((Late)arg1) -> TimeSlot :
Return the submitted time as a TimeSlot

class ecflow.Limit

Bases: `Boost.Python.instance`

limit provides a simple load management

i.e... by limiting the number of task s submitted by a server. Limits are typically defined at the suite level, or defined in a separate suite, so that they can be used by multiple suites. Once a limit is defined in a suite definition, you must also assign families/tasks to use this limit. See inlimit and `ecflow.InLimit`

Constructor:

```
Limit(name,value)
   string name: the name of the limit
   int   value: The value of the limit
```

Usage:

```
limit = Limit("fast", 10)
 ...
suite = Suite("s1",
              Limit("slow",10))  # create Limit in Node constructor
suite.add_limit(limit)           # add existing limit using function
```

decrement((Limit)arg1, (int)arg2, (str)arg3) None :

used for test only

increment((Limit)arg1, (int)arg2, (str)arg3) None :

used for test only

limit((Limit)arg1) int :

The max value of the limit as an integer

name((Limit)arg1) str :

Return the limit name as string

node_paths((Limit)arg1) list :

List of nodes(paths) that have consumed a limit

value((Limit)arg1) int :

The limit token value as an integer

class ecflow.Meter

Bases: `Boost.Python.instance`

meter s can be used to indicate proportional completion of task

They are able to trigger another job, which is waiting on this proportion. Can also be used to indicate progress of a job. Meters can be used in trigger and complete expression.

Constructor:

```
Meter(name,min,max,&lt;optional&gt;color_change)
   string name                : The meter name
   int min                    : The minimum and initial meter value
   int max                    : The maximum meter value. Must be greater than min value.
   int color_change&lt;optional&gt; : default = max, Must be between min-max, used in the GUI
```

Exceptions:

- raises IndexError when an invalid Meter is specified

Usage:

Using a meter requires:

- Defining a meter on a task:

```
meter = Meter("progress",0,100,100)
task.add_meter(meter)
```

- Updating the corresponding ecf script file with the meter child command:

```
ecflow_client --init=$$
for i in 10 20 30 40 50 60 80 100; do
    ecflow_client --meter=progress $i
    sleep 2 # or do some work
done
ecflow_client --complete
```

- Optionally addition in a trigger or complete expression for job control:

```
trigger task:progress ge 60
```

trigger and complete expression should *avoid* using equality i.e..:

```
trigger task:progress == 60
```

Due to network issues the meter event's may **not** arrive in sequential order hence the ecflow_server will ignore meter value's, which are less than the current value as a result triggers's which use meter equality may never evaluate

color_change((Meter)arg1) int :

returns the color change

empty((Meter)arg1) bool :

Return true if the Meter is empty. Used when returning a NULL Meter, from a find

max((Meter)arg1) int :

Return the Meters maximum value

min((Meter)arg1) int :

Return the Meters minimum value

name((Meter)arg1) str :

Return the Meters name as string

value((Meter)arg1) int :

Return meters current value

class ecflow.Node

Bases: `Boost.Python.instance`

A Node class is the abstract base class for Suite, Family and Task

Every Node instance has a name, and a path relative to a suite

add()

object add(tuple args, dict kwds) :
add(..) provides a way to append Nodes and attributes

This is best illustrated with an example:

```
defs = Defs().add(
    Suite("s1").add(
        Clock(1, 1, 2010, False),
        Autocancel(1, 10, True),
        Task("t1").add(
            Edit({"a":"12", "b":"bb"}, c="v",d="b"),
            Edit(g="d"),
            Edit(h=1),
            Event(1),
            Event(11,"event"),
            Meter("meter",0,10,10),
            Label("label","c"),
            Trigger("1==1"),
            Complete("1==1"),
            Limit("limit",10),Limit("limit2",10),
            InLimit("limitName","/limit",2),
            Defstatus(DState.complete),
            Today(0,30),Today("00:59"),Today("00:00 11:30 00:01"),
            Time(0,30),Time("00:59"),Time("00:00 11:30 00:01"),
            Day("sunday"),Day(Days.monday),
            Date(1,1,0),Date(28,2,1960),
            Autocancel(3)
            ),
        [ Family("f{}".format(i)) for i in range(1,6)]))
```

We can also use '+=' with a list here are a few examples:

```
defs = Defs();
defs += [ Suite("s2"),Edit({ "x1":"y", "aa1":"bb"}, a="v",b="b") ]
```

```
defs += [ Suite("s{}".format(i)) for i in range(1,6) ]
```

```
defs = Defs()
defs += [ Suite("suite").add(
             Task("x"),
             Family("f").add( [ Task("t{}".format(i)) for i in range(1,6)] ),
             Task("y"),
             [ Family("f{}".format(i)) for i in range(1,6) ],
             Edit(a="b"),
             [ Task("t{}".format(i)) for i in range(1,6) ],
             )]
```

It is also possible to use '+'

```
defs = Defs() + Suite("s1")
defs.s1 += Autocancel(1, 10, True)
defs.s1 += Task("t1") + Edit({ "e":1, "f":"bb"}) +\
           Event(1) + Event(11,"event") + Meter("meter",0,10,10) + Label("label","c") +
Trigger("1==1") +\
           Complete("1==1") + Limit("limit",10) + Limit("limit2",10) + InLimit("limitName","
/limit",2) +\
           Defstatus(DState.complete) + Today(0,30) + Today("00:59") + Today("00:00 11:30
00:01") +\
           Time(0,30) + Time("00:59") + Time("00:00 11:30 00:01") + Day("sunday") + Day
(Days.monday) +\
           Date(1,1,0) + Date(28,2,1960) + Autocancel(3)
```

Warning

We can only use '+' when the left most object is a node, i.e.. Task('t1') in this case

add_autocancel((Node)arg1, (int)arg2) Node :

Add a *autocancel* attribute. See `ecflow.Autocancel`

This will delete the node on completion. The deletion may be delayed by an amount of time in hours and minutes or expressed as days Node deletion is not immediate. The nodes are checked once a minute and expired auto cancel nodes are deleted A node may only have one auto cancel attribute

Exception:

- Throws a RuntimeError if more than one auto cancel is added

Usage:

```
t1 = Task("t1")
t1.add_autocancel( Autocancel(20,10,False) )  # hour,min, relative
t2 = Task("t2")
t2.add_autocancel( 3 )                        # 3 days
t3 = Task("t3")
t3.add_autocancel( 20,10,True )               # hour,minutes,relative
t4 = Task("t4")
t4.add_autocancel( TimeSlot(20,10),True )     # hour,minutes,relative

# we can also create a Autocancel in the Task constructor like any other attribute
t2 = Task("t2",
          Autocancel(20,10,False))
```

add_autocancel((Node)arg1, (int)arg2, (int)arg3, (bool)arg4) -> Node

add_autocancel((Node)arg1, (TimeSlot)arg2, (bool)arg3) -> Node

add_autocancel((Node)arg1, (Autocancel)arg2) -> Node

add_complete((Node)arg1, (str)arg2) Node :

Add a trigger or complete expression.Also see `ecflow.Trigger`

This defines a dependency for a node. There can only be one trigger or complete expression dependency per node. A node with a trigger can only be activated when the trigger has expired. A trigger holds a node as long as the expression returns false.

Exception:

- Will throw RuntimeError if multiple trigger or complete expression are added
- Will throw RuntimeError if first expression is added as 'AND' or 'OR' expression Like wise second and subsequent expression must have 'AND' or 'OR' booleans set

Usage:

Note we can not make multiple add_trigger(..) calls on the same task! to add a simple trigger:

```
task1.add_trigger( "t2 == active" )
task2.add_trigger( "t1 == complete or t4 == complete" )
task3.add_trigger( "t5 == active" )
```

Long expression can be broken up using add_part_trigger:

```
task2.add_part_trigger( "t1 == complete or t4 == complete")
task2.add_part_trigger( "t5 == active",True)  # True means  AND
task2.add_part_trigger( "t7 == active",False) # False means OR
```

The trigger for task2 is equivalent to: 't1 == complete or t4 == complete and t5 == active or t7 == active'

add_complete((Node)arg1, (Expression)arg2) -> Node

add_cron((Node)arg1, (Cron)arg2) Node :

Add a cron time dependency. See `ecflow.Cron`

Usage:

```
start = TimeSlot(0,0)
finish = TimeSlot(23,0)
incr = TimeSlot(0,30)
time_series = TimeSeries( start, finish, incr, True)
cron = Cron()
cron.set_week_days( [0,1,2,3,4,5,6] )
cron.set_days_of_month( [1,2,3,4,5,6] )
cron.set_months( [1,2,3,4,5,6] )
cron.set_time_series( time_series )
t1 = Task("t1")
t1.add_cron( cron )

# we can also create a Cron in the Task constructor like any other attribute
t2 = Task("t2",
          Cron("+00:00 23:00 00:30",days_of_week=[0,1,2,3,4,5,6],days_of_month=[1,2,3,4,5,6],
months=[1,2,3,4,5,6]))
```

add_date((Node)arg1, (int)arg2, (int)arg3, (int)arg4) Node :

Add a date time dependency. See `ecflow.Date`

A value of zero for day,month,year means every day, every month, every year

Exception:

- Throws RuntimeError if an invalid date is added

Usage:

```
t1 = Task("t1",
          Date("1.*.*"),
          Date(1,1,2010)))    # Create Date in place

t1.add_date( Date(1,1,2010) ) # day,month,year
t1.add_date( 2,1,2010)        # day,month,year
t1.add_date( 1,0,0)           # day,month,year, the first of each month for every year
```

add_date((Node)arg1, (Date)arg2) -> Node

add_day((Node)arg1, (Days)arg2) Node :

Add a day time dependency. See `ecflow.Day`

Usage:

```
t1 = Task("t1",
          Day("sunday"))  # Create Day on Task creation

t1.add_day( Day(Days.sunday) )
t1.add_day( Days.monday)
t1.add_day( "tuesday" )
```

add_day((Node)arg1, (str)arg2) -> Node

add_day((Node)arg1, (Day)arg2) -> Node

add_defstatus((Node)arg1, (DState)arg2) Node :

Set the default status(defstatus) of node at begin or re queue. See `ecflow.Defstatus`

A defstatus is useful in preventing suites from running automatically once begun, or in setting Task's complete so they can be run selectively

Usage:

```
t1 = Task("t1") + Defstatus("complete")
t2 = Task("t2").add_defstatus( DState.suspended )

# we can also create a Defstatus in the Task constructor like any other attribute
t2 = Task("t3",
          Defstatus("complete"))
```

add_defstatus((Node)arg1, (Defstatus)arg2) -> Node :

Set the default status(defstatus) of node at begin or re queue. See `ecflow.Defstatus`

A defstatus is useful in preventing suites from running automatically once begun, or in setting Task's complete so they can be run selectively

Usage:

```
t1 = Task("t1") + Defstatus("complete")
t2 = Task("t2").add_defstatus( DState.suspended )

# we can also create a Defstatus in the Task constructor like any other attribute
t2 = Task("t3",
          Defstatus("complete"))
```

add_event((Node)arg1, (Event)arg2) Node :

Add a event. See `ecflow.Event`

Events can be referenced in trigger and complete expression s

Exception:

- Throws RuntimeError if a duplicate is added

Usage:

```
t1 = Task("t1",
          Event(12),
          Event(11,"eventx"))             # Create events on Task creation

t1.add_event( Event(10) )                 # Create with function on Task
t1.add_event( Event(11,"Eventname") )
t1.add_event( 12 )
t1.add_event( 13, "name")
```

To reference event 'flag' in a trigger:

```
t1.add_event("flag")
t2 = Task("t2",
          Trigger("t1:flag == set"))
```

add_event((Node)arg1, (int)arg2) -> Node

add_event((Node)arg1, (int)arg2, (str)arg3) -> Node

add_event((Node)arg1, (str)arg2) -> Node

add_inlimit((Node)arg1, (str)limit_name[, (str)path_to_node_containing_limit='', (int)tokens=1]) Node :

Adds a inlimit to a node. See `ecflow.InLimit`

InLimit reference a limit/`ecflow.Limit`. Duplicate InLimits are not allowed

Exception:

- Throws RuntimeError if a duplicate is added

Usage:

```
task2.add_inlimit( InLimit("limitName","/s1/f1",2) )
task2.add_inlimit( "limitName","/s1/f1",2 )
```

add_inlimit((Node)arg1, (InLimit)arg2) -> Node

add_label((Node)arg1, (str)arg2, (str)arg3) Node :

Adds a label to a node. See `ecflow.Label`

Labels can be updated from the jobs files, via child command

Exception:

- Throws RuntimeError if a duplicate label name is added

Usage:

```
task.add_label( Label("TEA","/me/"))
task.add_label( "Joe","/me/")
```

The corresponding child command in the .ecf script file might be:

```
ecflow_client --label=TEA time
ecflow_client --label=Joe ninety
```

add_label((Node)arg1, (Label)arg2) -> Node

add_late((Node)arg1, (Late)arg2) Node :

Add a late attribute. See `ecflow.Late`

Exception:

- Throws a RuntimeError if more than one late is added

Usage:

```
late = Late()
late.submitted( 20,10 )     # hour,minute
late.active(    20,10 )     # hour,minute
late.complete(  20,10,True) # hour,minute,relative
t1 = Task("t1")
t1.add_late( late )

# we can also create a Late in the Task constructor like any other attribute
t2 = Task("t2",
          Late(submitted="20:10",active="20:10",complete="+20:10"))
```

add_limit((Node)arg1, (str)arg2, (int)arg3) Node :

Adds a limit to a node for simple load management. See `ecflow.Limit`

Multiple limits can be added, however the limit name must be unique. For a node to be in a limit, a inlimit must be used.

Exception:

- Throws RuntimeError if a duplicate limit name is added

Usage:

```
family.add_limit( Limit("load",12) )
family.add_limit( "load",12 )
```

add_limit((Node)arg1, (Limit)arg2) -> Node

add_meter((Node)arg1, (Meter)arg2) Node :

Add a meter. See `ecflow.Meter`

Meters can be referenced in trigger and complete expression s

Exception:

- Throws RuntimeError if a duplicate is added

Usage:

```
t1 = Task("t1",
          Meter("met",0,50))                    # create Meter on Task creation
t1.add_meter( Meter("metername",0,100,50) )  # create Meter using function
t1.add_meter( "meter",0,200)
```

To reference in a trigger:

```
t2 = Task("t2")
t2.add_trigger("t1:meter &gt;= 10")
```

add_meter((Node)arg1, (str)arg2, (int)arg3, (int)arg4 [, (int)arg5]) -> Node

add_part_complete((Node)arg1, (PartExpression)arg2) Node :

Add a trigger or complete expression.Also see `ecflow.Trigger`

This defines a dependency for a node. There can only be one trigger or complete expression dependency per node. A node with a trigger can only be activated when the trigger has expired. A trigger holds a node as long as the expression returns false.

Exception:

- Will throw RuntimeError if multiple trigger or complete expression are added
- Will throw RuntimeError if first expression is added as 'AND' or 'OR' expression Like wise second and subsequent expression must have 'AND' or 'OR' booleans set

Usage:

Note we can not make multiple add_trigger(..) calls on the same task! to add a simple trigger:

```
task1.add_trigger( "t2 == active" )
task2.add_trigger( "t1 == complete or t4 == complete" )
task3.add_trigger( "t5 == active" )
```

Long expression can be broken up using add_part_trigger:

```
task2.add_part_trigger( "t1 == complete or t4 == complete")
task2.add_part_trigger( "t5 == active",True)  # True means  AND
task2.add_part_trigger( "t7 == active",False) # False means OR
```

The trigger for task2 is equivalent to: 't1 == complete or t4 == complete and t5 == active or t7 == active'

add_part_complete((Node)arg1, (str)arg2) -> Node

add_part_complete((Node)arg1, (str)arg2, (bool)arg3) -> Node

add_part_trigger((Node)arg1, (PartExpression)arg2) Node :

Add a trigger or complete expression.Also see `ecflow.Trigger`

This defines a dependency for a node. There can only be one trigger or complete expression dependency per node. A node with a trigger can only be activated when the trigger has expired. A trigger holds a node as long as the expression returns false.

Exception:

- Will throw RuntimeError if multiple trigger or complete expression are added
- Will throw RuntimeError if first expression is added as 'AND' or 'OR' expression Like wise second and subsequent expression must have 'AND' or 'OR' booleans set

Usage:

Note we can not make multiple add_trigger(..) calls on the same task! to add a simple trigger:

```
task1.add_trigger( "t2 == active" )
task2.add_trigger( "t1 == complete or t4 == complete" )
task3.add_trigger( "t5 == active" )
```

Long expression can be broken up using add_part_trigger:

```
task2.add_part_trigger( "t1 == complete or t4 == complete")
task2.add_part_trigger( "t5 == active",True)  # True means  AND
task2.add_part_trigger( "t7 == active",False) # False means OR
```

The trigger for task2 is equivalent to: 't1 == complete or t4 == complete and t5 == active or t7 == active'

add_part_trigger((Node)arg1, (str)arg2) -> Node

add_part_trigger((Node)arg1, (str)arg2, (bool)arg3) -> Node

add_repeat((Node)arg1, (RepeatDate)arg2) Node :

Add a RepeatDate attribute. See `ecflow.RepeatDate`

A node can only have one repeat

Exception:

- Throws a RuntimeError if more than one repeat is added

Usage:

```
t1 = Task("t1")
t1.add_repeat( RepeatDate("testDate",20100111,20100115) )

# we can also create a repeat in Task constructor like any other attribute
t2 = Task("t2",
          RepeatDate("testDate",20100111,20100115))
```

add_repeat((Node)arg1, (RepeatInteger)arg2) -> Node :

Add a RepeatInteger attribute. See `ecflow.RepeatInteger`

A node can only have one repeat

Exception:

- Throws a RuntimeError if more than one repeat is added

Usage:

```
t1 = Task("t1")
t1.add_repeat( RepeatInteger("testInteger",0,100,2) )

# we can also create a repeat in Task constructor like any other attribute
t2 = Task("t2",
          RepeatInteger("testInteger",0,100,2))
```

add_repeat((Node)arg1, (RepeatString)arg2) -> Node :
Add a RepeatString attribute. See `ecflow.RepeatString`

A node can only have one repeat

Exception:

- Throws a RuntimeError if more than one repeat is added

Usage:

```
t1 = Task("t1")
t1.add_repeat( RepeatString("test_string",["a", "b", "c" ] ) )

# we can also create a repeat in Task constructor like any other attribute
t2 = Task("t2",
          RepeatString("test_string",["a", "b", "c" ] ) )
```

add_repeat((Node)arg1, (RepeatEnumerated)arg2) -> Node :
Add a RepeatEnumerated attribute. See `ecflow.RepeatEnumerated`

A node can only have one repeat

Exception:

- Throws a RuntimeError if more than one repeat is added

Usage:

```
t1 = Task("t1")
t1.add_repeat( RepeatEnumerated("test_string", ["red", "green", "blue" ] ) )

# we can also create a repeat in Task constructor like any other attribute
t2 = Task("t2",
          RepeatEnumerated("test_string", ["red", "green", "blue" ] ) )
```

add_repeat((Node)arg1, (RepeatDay)arg2) -> Node :
Add a RepeatDay attribute. See `ecflow.RepeatDay`

A node can only have one repeat

Exception:

- Throws a RuntimeError if more than one repeat is added

Usage:

```
t2 = Task("t2",
          RepeatDay(1))
```

add_time((Node)arg1, (int)arg2, (int)arg3) Node :

Add a time dependency. See `ecflow.Time`

Usage:

```
t1 = Task("t1", Time("+00:30 20:00 01:00")) # Create Time in Task constructor
t1.add_time( "00:30" )
t1.add_time( "+00:30" )
t1.add_time( "+00:30 20:00 01:00" )
t1.add_time( Time( 0,10 ))      # hour,min,relative =false
t1.add_time( Time( 0,12,True )) # hour,min,relative
t1.add_time( Time(TimeSlot(20,20),False))
t1.add_time( 0,1 ))              # hour,min,relative=false
t1.add_time( 0,3,False ))        # hour,min,relative=false
start = TimeSlot(0,0)
finish = TimeSlot(23,0)
incr = TimeSlot(0,30)
ts = TimeSeries( start, finish, incr, True)
task2.add_time( Time(ts) )
```

add_time((Node)arg1, (int)arg2, (int)arg3, (bool)arg4) -> Node

add_time((Node)arg1, (str)arg2) -> Node

add_time((Node)arg1, (Time)arg2) -> Node

add_today((Node)arg1, (int)arg2, (int)arg3) Node :

Add a today time dependency. See `ecflow.Today`

Usage:

```
t1 = Task("t1",
          Today("+00:30 20:00 01:00")) # Create Today in Task constructor

t1.add_today( "00:30" )
t1.add_today( "+00:30" )
t1.add_today( "+00:30 20:00 01:00" )
t1.add_today( Today( 0,10 ))      # hour,min,relative =false
t1.add_today( Today( 0,12,True )) # hour,min,relative
t1.add_today( Today(TimeSlot(20,20),False))
t1.add_today( 0,1 ))              # hour,min,relative=false
t1.add_today( 0,3,False ))        # hour,min,relative=false
start = TimeSlot(0,0)
finish = TimeSlot(23,0)
incr = TimeSlot(0,30)
ts = TimeSeries( start, finish, incr, True)
task2.add_today( Today(ts) )
```

add_today((Node)arg1, (int)arg2, (int)arg3, (bool)arg4) -> Node

add_today((Node)arg1, (str)arg2) -> Node

add_today((Node)arg1, (Today)arg2) -> Node

add_trigger((Node)arg1, (str)arg2) Node :

Add a trigger or complete expression.Also see `ecflow.Trigger`

This defines a dependency for a node. There can only be one trigger or complete expression dependency per node. A node with a trigger can only be activated when the trigger has expired. A trigger holds a node as long as the expression returns false.

Exception:

- Will throw RuntimeError if multiple trigger or complete expression are added
- Will throw RuntimeError if first expression is added as ‘AND’ or ‘OR’ expression Like wise second and subsequent expression must have ‘AND’ or ‘OR’ booleans set

Usage:

Note we can not make multiple add_trigger(..) calls on the same task! to add a simple trigger:

```
task1.add_trigger( "t2 == active" )
task2.add_trigger( "t1 == complete or t4 == complete" )
task3.add_trigger( "t5 == active" )
```

Long expression can be broken up using add_part_trigger:

```
task2.add_part_trigger( "t1 == complete or t4 == complete")
task2.add_part_trigger( "t5 == active",True)  # True means  AND
task2.add_part_trigger( "t7 == active",False) # False means OR
```

The trigger for task2 is equivalent to: 't1 == complete or t4 == complete and t5 == active or t7 == active'

add_trigger((Node)arg1, (Expression)arg2) -> Node

add_variable((Node)arg1, (str)arg2, (str)arg3) Node :

Adds a name value variable. Also see `ecflow.Edit`

This defines a variable for use in variable substitution in a ecf script file. There can be any number of variables. The variables are names inside a pair of '%' characters in an ecf script. The name are case sensitive. Special character in the value, must be placed inside single quotes if misinterpretation is to be avoided. The value of the variable replaces the variable name in the ecf script at *job creation* time. The variable names for any given node must be unique. If duplicates are added then the the last value added is kept.

Exception:

- Writes warning to standard output, if a duplicate variable name is added

Usage:

```
task.add_variable( Variable("ECF_HOME","/tmp/"))
task.add_variable( "TMPDIR","/tmp/")
task.add_variable( "COUNT",2)
a_dict = { "name":"value", "name2":"value2", "name3":"value3" }
task.add_variable(a_dict)
```

add_variable((Node)arg1, (str)arg2, (int)arg3) -> Node

add_variable((Node)arg1, (Variable)arg2) -> Node

add_variable((Node)arg1, (dict)arg2) -> Node

add_verify((Node)arg1, (Verify)arg2) None :

Add a Verify attribute.

Used in python simulation used to assert that a particular state was reached. t2 = Task('t2',
Verify(State.complete, 6)) # verify task completes 6 times during simulation

add_zombie((Node)arg1, (ZombieAttr)arg2) Node :

The zombie attribute defines how a zombie should be handled in an automated fashion

Very careful consideration should be taken before this attribute is added as it may hide a genuine problem. It can be added to any node. But is best defined at the suite or family level. If there is no zombie attribute the default behaviour is to block the init,complete,abort child command. and *fob* the event,label,and meter child command This attribute allows the server to make a automated response. Please see: `ecflow.ZombieType`, `ecflow.ChildCmdType`, `ecflow.ZombieUserActionType`

Constructor:

```
ZombieAttr(ZombieType,ChildCmdTypes, ZombieUserActionType, lifetime)
   ZombieType            : Must be one of ZombieType.ecf, ZombieType.path, ZombieType.user
   ChildCmdType          : A list(ChildCmdType) of Child commands. Can be left empty in
                           which case the action affect all child commands
   ZombieUserActionType  : One of [ fob, fail, block, remove, adopt ]
   int lifetime&lt;optional&gt;: Defines the life time in seconds of the zombie in the server.
                           On expiration, zombie is removed automatically
```

Usage:

```
# Add a zombie attribute so that child commands(i.e.. ecflow_client --init)
# will fail the job if it is a zombie process.
s1 = Suite("s1")
child_list = [ ChildCmdType.init, ChildCmdType.complete, ChildCmdType.abort ]
s1.add_zombie( ZombieAttr(ZombieType.ecf, child_list, ZombieUserActionType.fob))

# create the zombie as part of the node constructor
s1 = Suite("s1",
           ZombieAttr(ZombieType.ecf, child_list, ZombieUserActionType.fail))
```

change_complete((Node)arg1, (str)arg2) None

change_trigger((Node)arg1, (str)arg2) None

delete_complete((Node)arg1) None

delete_cron((Node)arg1, (str)arg2) None

delete_cron((Node)arg1, (Cron)arg2) -> None

delete_date((Node)arg1, (str)arg2) None

delete_date((Node)arg1, (Date)arg2) -> None

delete_day((Node)arg1, (str)arg2) None

delete_day((Node)arg1, (Day)arg2) -> None

delete_event((Node)arg1, (str)arg2) None

delete_inlimit((Node)arg1, (str)arg2) None

delete_label((Node)arg1, (str)arg2) None

delete_limit((Node)arg1, (str)arg2) None

delete_meter((Node)arg1, (str)arg2) None

delete_repeat((Node)arg1) None

delete_time((Node)arg1, (str)arg2) None

delete_time((Node)arg1, (Time)arg2) -> None

delete_today((Node)arg1, (str)arg2) None

delete_today((Node)arg1, (Today)arg2) -> None

delete_trigger((Node)arg1) None

delete_variable((Node)arg1, (str)arg2) None

delete_zombie((Node)arg1, (str)arg2) None

delete_zombie((Node)arg1, (ZombieType)arg2) -> None

evaluate_complete((Node)arg1) bool :

evaluate complete expression

evaluate_trigger((Node)arg1) bool :

evaluate trigger expression

find_event((Node)arg1, (str)arg2) Event :

Find the event on the node only. Returns a object

find_gen_variable((Node)arg1, (str)arg2) Variable :

Find generated variable on the node only. Returns a object

find_label((Node)arg1, (str)arg2) Label :

Find the label on the node only. Returns a object

find_limit((Node)arg1, (str)arg2) Limit :

Find the limit on the node only. returns a limit ptr

find_meter((Node)arg1, (str)arg2) Meter :

Find the meter on the node only. Returns a object

find_node_up_the_tree((Node)arg1, (str)arg2) Node :

Search immediate node, then up the node hierarchy

find_parent_variable((Node)arg1, (str)arg2) Variable :

Find user variable variable up the parent hierarchy. Returns a object

find_variable((Node)arg1, (str)arg2) Variable :

Find user variable on the node only. Returns a object

get_abs_node_path((Node)arg1) str :

returns a string which holds the path to the node

get_all_nodes((Node)arg1) NodeVec :

Returns all the child nodes

get_autocancel((Node)arg1) Autocancel

get_complete((Node)arg1) Expression

get_defs((Node)arg1) Defs

get_defstatus((Node)arg1) DState

get_dstate((Node)arg1) DState :

Returns the state of node. This will include suspended state

get_flag((Node)arg1) Flag :

Return additional state associated with a node.

get_generated_variables((Node)arg1, (VariableList)arg2) None :

returns a list of generated variables. Use ecflow.VariableList as return argument

get_late((Node)arg1) Late

get_parent((Node)arg1) Node

get_repeat((Node)arg1) Repeat

get_state((Node)arg1) State :

Returns the state of the node. This excludes the suspended state

get_state_change_time((Node)arg1[, (str)format='iso_extended']) str :

Returns the time of the last state change as a string. Default format is iso_extended, (iso_extended, iso, simple)

get_trigger((Node)arg1) Expression

has_time_dependencies((Node)arg1) bool

is_suspended((Node)arg1) bool :

Returns true if the node is in a suspended state

name((Node)arg1) str

remove((Node)arg1) Node :

Remove the node from its parent. and returns it

replace_on_server((Node)arg1[, (bool)suspend_node_first=True[, (bool)force=True]]) None :

replace node on the server.

replace_on_server((Node)arg1, (str)arg2, (str)arg3 [, (bool)suspend_node_first=True [, (bool)force=True]]) -> None :
    replace node on the server.
replace_on_server((Node)arg1, (str)arg2 [, (bool)suspend_node_first=True [, (bool)force=True]]) -> None :
    replace node on the server.

sort_attributes((Node)arg1, (AttrType)attribute_type[, (bool)recursive=True]) None

sort_attributes((Node)arg1, (str)attribute_type [, (bool)recursive=True]) -> None

update_generated_variables((Node)arg1) None

```
crons
Returns a list of cron s
dates
Returns a list of date s
days
Returns a list of day s
events
Returns a list of event s
inlimits
Returns a list of inlimit s
labels
Returns a list of label s
limits
Returns a list of limit s
meters
Returns a list of meter s
times
Returns a list of time s
todays
Returns a list of today s
variables
Returns a list of user defined variable s
verifies
Returns a list of Verify&amp;#8217;s
zombies
Returns a list of zombie s
```

class ecflow.NodeContainer

Bases: `ecflow.Node`

NodeContainer is the abstract base class for a Suite and Family

A NodeContainer can have Families and Tasks as children

add_family((NodeContainer)arg1, (str)arg2) Family :

Add a family. See `ecflow.Family`.

Multiple families can be added. However family names must be unique. for a given parent. Families can be hierarchical.

Exception:

- Throws RuntimeError if a duplicate is added

Usage:

```
suite = Suite("suite")          # create a suite
f1 = Family("f1")               # create a family
suite.add_family(f1)            # add family to suite
f2 = suite.add_family("f2")     # create a family and add to suite
```

add_family((NodeContainer)arg1, (Family)arg2) -> Family

add_task((NodeContainer)arg1, (str)arg2) Task :

Add a task. See `ecflow.Task`

Multiple Tasks can be added. However Task names must be unique, for a given parent. Task can be added to Familiy's or Suites.

Exception:

- Throws RuntimeError if a duplicate is added

Usage:

```
f1 = Family("f1")      # create a family
t1 = Task("t1")          # create a task
f1.add_task(t1)          # add task to family
t2 = f1.add_task("t2") # create task "t2" and add to family
```

add_task((NodeContainer)arg1, (Task)arg2) -> Task

find_family((NodeContainer)arg1, (str)arg2) Family :

Find a family given a name

find_task((NodeContainer)arg1, (str)arg2) Task :

Find a task given a name

```
nodes
Returns a list of Node&amp;#8217;s
```

class ecflow.NodeVec

Bases: `Boost.Python.instance`

Hold a list of Nodes (i.e.. suite, family or task s)

append((NodeVec)arg1, (object)arg2) None

extend((NodeVec)arg1, (object)arg2) None

class ecflow.PartExpression

Bases: `Boost.Python.instance`

PartExpression holds part of a trigger or complete expression.

Expressions can contain references to event, meter s, user variables, repeat variables and generated variables. The part expression allows us to split a large trigger or complete expression into smaller ones

Constructor:

```
PartExpression(exp )
    string   exp: This represents the *first* expression

PartExpression(exp, bool and_expr)
    string   exp: This represents the expression
    bool and_exp: If true the expression is to be anded, with a previously added expression
                  If false the expression is to be "ored", with a previously added expression
```

Usage: To add simple expression this class can be by-passed, i.e... can use:

```
task = Task("t1")
task.add_trigger( "t2 == active" )
task.add_complete( "t2 == complete" )
```

To add large triggers and complete expression:

```
exp1 = PartExpression("t1 == complete")
# a simple expression can be added as a string
....
task2.add_part_trigger( PartExpression("t1 == complete or t4 == complete") )
task2.add_part_trigger( PartExpression("t5 == active",True) )    # anded with first expression
task2.add_part_trigger( PartExpression("t7 == active",False) )   # or"ed with last expression added
```

The trigger for task2 is equivalent to 't1 == complete or t4 == complete and t5 == active or t7 == active'

and_expr((PartExpression)arg1) bool

get_expression((PartExpression)arg1) str :

returns the part expression as a string

or_expr((PartExpression)arg1) bool

class ecflow.PrintStyle

Bases: `Boost.Python.instance`

Singleton used to control the print Style. See `ecflow.Style`

Usage:

```
old_style = PrintStyle.get_style()
PrintStyle.set_style(PrintStyle.STATE)
...
print(defs)                    # show the node state
PrintStyle.set_style(old_style) # reset previous style
```

static get_style() Style :

Returns the style, static method

static set_style((Style)arg1) None :

Set the style, static method

class ecflow.Repeat

Bases: `Boost.Python.instance`

Represents one of RepeatString,RepeatEnumerated,RepeatInteger,RepeatDate,RepeatDay

empty((Repeat)arg1) bool :

Return true if the repeat is empty.

end((Repeat)arg1) int :

The last value of the repeat, as an integer

name((Repeat)arg1) str :

The repeat name, can be referenced in trigger expressions

start((Repeat)arg1) int :

The start value of the repeat, as an integer

step((Repeat)arg1) int :

The increment for the repeat, as an integer

value((Repeat)arg1) int :

The current value of the repeat as an integer

class ecflow.RepeatDate

Bases: `Boost.Python.instance`

Allows a node to be repeated using a yyyymmdd format

A node can only have one repeat. The repeat name can be referenced in trigger expressions.

Constructor:

```
RepeatDate(variable,start,end,delta)
   string variable:     The name of the repeat. The current date can referenced in
                        in trigger expressions using the variable name
   int start:           Start date, must have format: yyyymmdd
   int end:             End date, must have format: yyyymmdd
   int delta&lt;optional&gt;: default = 1, Always in days. The increment used to update the date
```

Exception:

- Throws a RuntimeError if start/end are not valid dates

Usage:

```
rep = RepeatDate("YMD", 20050130, 20050203 )
rep = RepeatDate("YMD", 20050130, 20050203, 2 )
t = Task("t1",
         RepeatDate("YMD", 20050130, 20050203 ) )
```

end((RepeatDate)arg1) int :

Return the end date as an integer in yyyymmdd format

name((RepeatDate)arg1) str :

Return the name of the repeat.

start((RepeatDate)arg1) int :

Return the start date as an integer in yyyymmdd format

step((RepeatDate)arg1) int :

Return the step increment. This is used to update the repeat, until end date is reached

class ecflow.RepeatDay

Bases: `Boost.Python.instance`

A repeat that is infinite.

A node can only have one repeat.

Constructor:

```
RepeatDay(step)
   int step:     The step.
```

Usage:

```
t = Task("t1",
          RepeatDay( 1 ))
```

class ecflow.RepeatEnumerated

Bases: `Boost.Python.instance`

Allows a node to be repeated using a enumerated list.

A node can only have one repeat. The repeat can be referenced in trigger expressions.

Constructor:

```
RepeatEnumerated(variable,list)
   string variable:     The name of the repeat. The current enumeration index can be
                        referenced in trigger expressions using the variable name
   vector list:         The list of enumerations
```

Usage:

```
t = Task("t1",
         RepeatEnumerated("COLOR", [ "red", "green", "blue" ] ))
```

end((RepeatEnumerated)arg1) int

name((RepeatEnumerated)arg1) str :

Return the name of the repeat.

start((RepeatEnumerated)arg1) int

step((RepeatEnumerated)arg1) int

class ecflow.RepeatInteger

Bases: `Boost.Python.instance`

Allows a node to be repeated using a integer range.

A node can only have one repeat. The repeat can be referenced in trigger expressions.

Constructor:

```
RepeatInteger(variable,start,end,step)
   string variable:     The name of the repeat. The current integer value can be
                        referenced in trigger expressions using the variable name
   int start:           Start integer value
   int end:             End end integer value
   int step&lt;optional&gt;:  Default = 1, The step amount
```

Usage:

```
t = Task("t1",
         RepeatInteger("HOUR", 6, 24, 6 ))
```

end((RepeatInteger)arg1) int

name((RepeatInteger)arg1) str :

Return the name of the repeat.

start((RepeatInteger)arg1) int

step((RepeatInteger)arg1) int

class ecflow.RepeatString

Bases: `Boost.Python.instance`

Allows a node to be repeated using a string list.

A node can only have one repeat. The repeat can be referenced in trigger expressions.

Constructor:

```
RepeatString(variable,list)
   string variable:     The name of the repeat. The current index of the string list can be
                        referenced in trigger expressions using the variable name
   vector list:         The list of enumerations
```

Usage:

```
t = Task("t1",
         RepeatString("COLOR", [ "red", "green", "blue" ] ))
```

end((RepeatString)arg1) int

name((RepeatString)arg1) str :

Return the name of the repeat.

start((RepeatString)arg1) int

step((RepeatString)arg1) int

class ecflow.SState

Bases: `Boost.Python.enum`

A SState holds the ecflow_server state

See server states

```
HALTED = ecflow.SState.HALTED
RUNNING = ecflow.SState.RUNNING
SHUTDOWN = ecflow.SState.SHUTDOWN
names = {'HALTED': ecflow.SState.HALTED, 'RUNNING': ecflow.SState.RUNNING, 'SHUTDOWN': ecflow.SState.SHUTDOWN}
values = {0: ecflow.SState.HALTED, 1: ecflow.SState.SHUTDOWN, 2: ecflow.SState.RUNNING}
```

class ecflow.State

Bases: `Boost.Python.enum`

Each node can have a status, which reflects the life cycle of a node.

It varies as follows:

- When the definition file is loaded into the ecflow_server the task status is unknown
- After begin command the task s are either queued, complete, aborted or suspended , a suspended task means that the task is really queued but it must be resumed by the user first before it can be submitted. See `ecflow.DState`
- Once the dependencies are resolved a task is submitted and placed into the submitted state, however if the submission fails, the task is placed in a aborted state.
- On a successful submission the task is placed into the active state
- Before a job ends, it may send other message to the server such as: Set an event, Change a meter, Change a label, send a message to log file

Jobs end by becoming either complete or aborted

```
aborted = ecflow.State.aborted
active = ecflow.State.active
complete = ecflow.State.complete
names = {'complete': ecflow.State.complete, 'unknown': ecflow.State.unknown, 'submitted': ecflow.State.
submitted, 'aborted': ecflow.State.aborted, 'active': ecflow.State.active, 'queued': ecflow.State.queued}
queued = ecflow.State.queued
submitted = ecflow.State.submitted
unknown = ecflow.State.unknown
values = {0: ecflow.State.unknown, 1: ecflow.State.complete, 2: ecflow.State.queued, 3: ecflow.State.aborted,
4: ecflow.State.submitted, 5: ecflow.State.active}
```

class ecflow.Style

Bases: `Boost.Python.enum`

Style is used to control printing output for the definition

- DEFS: This style outputs the definition file in a format that is parse-able.
  and can be re-loaded back into the server. Externs are automatically added. This excludes the edit history.
- STATE: The output includes additional state information for debug
  This excludes the edit history
- MIGRATE: Output includes structure and state, allow migration to future ecflow versions
  This includes edit history. If file is reloaded no checking is done

The following shows a summary of the features associated with each choice

Functionality	DEFS	STATE	MIGRATE
Auto generate externs	Yes	Yes	No
Checking on reload	Yes	Yes	No
Edit History	No	No	Yes
Show trigger AST	No	Yes	No

```
DEFS = ecflow.Style.DEFS
MIGRATE = ecflow.Style.MIGRATE
NOTHING = ecflow.Style.NOTHING
STATE = ecflow.Style.STATE
names = {'NOTHING': ecflow.Style.NOTHING, 'DEFS': ecflow.Style.DEFS, 'STATE': ecflow.Style.STATE, 'MIGRATE':
ecflow.Style.MIGRATE}
values = {0: ecflow.Style.NOTHING, 1: ecflow.Style.DEFS, 2: ecflow.Style.STATE, 3: ecflow.Style.MIGRATE}
```

class ecflow.Submittable

Bases: `ecflow.Node`

Submittable is the abstract base class for a Task and Alias

It provides a process id, password and try number

get_aborted_reason((Submittable)arg1) str :

If node was aborted and a reason was provided, return the string

get_int_try_no((Submittable)arg1) int :

The current try number as integer.

get_jobs_password((Submittable)arg1) str :

The password. This generated by server

get_process_or_remote_id((Submittable)arg1) str :

The process or remote id of the running job

get_try_no((Submittable)arg1) str :

The current try number as a string.

class ecflow.Suite

Bases: `ecflow.NodeContainer`

A suite is a collection of Families,Tasks,Variables, repeat and clock definitions

Suite is the only node that can be started using the begin API. There are several ways of adding a suite, see example below and `ecflow.Defs.add_suite`

Constructor:

```
Suite(name, Nodes | attributes)
    string name : The Suite name. name must consist of alpha numeric characters or
                  underscore or dot. The first character can not be a dot, as this
                  will interfere with trigger expressions. Case is significant
    Nodes | Attributes:(optional)
```

Exception:

- Throws a RuntimeError if the name is not valid
- Throws a RuntimeError if duplicate suite names added

Usage:

```
defs = Defs()                  # create a empty definition. Root of all Suites
suite = Suite("suite_1")       # create a stand alone suite
defs.add_suite(suite)          # add suite to definition
suite2 = defs.add_suite("s2")  # create a suite and add it to the defs

defs = Defs(
         Suite("s1",
            Family("f1",
               Task("t1"))))   # create in in-place
```

add_clock((Suite)arg1, (Clock)arg2) Suite

add_end_clock((Suite)arg1, (Clock)arg2) Suite :

End clock, used to mark end of simulation

begun((Suite)arg1) bool :

Returns true if the suite has begun, false otherwise

get_clock((Suite)arg1) Clock :

Returns the suite clock

get_end_clock((Suite)arg1) Clock :

Return the suite's end clock. Can be NULL

class ecflow.SuiteVec

Bases: `Boost.Python.instance`

Hold a list of suite nodes's

append((SuiteVec)arg1, (object)arg2) None

extend((SuiteVec)arg1, (object)arg2) None

class ecflow.Task

Bases: `ecflow.Submittable`

Creates a task node. Task is a child of a `ecflow.Suite` or `ecflow.Family` node.

Multiple Tasks can be added, however the task names must be unique for a given parent. Note case is significant. Only Tasks can be submitted. A job inside a Task ecf script (i.e.. .ecf file) should generally be re-entrant since a Task may be automatically submitted more than once if it aborts. There are several ways of adding a task, see examples below

Constructor:

```
Task(name, Attributes)
   string name : The Task name.Name must consist of alpha numeric characters or
                 underscore or dot. First character can not be a dot.
                 Case is significant
   attributes: optional, i.e.. like Meter, Event, Trigger etc
```

Exception:

- Throws a RuntimeError if the name is not valid
- Throws a RuntimeError if a duplicate Task is added

Usage:

```
task = Task("t1")            # create a stand alone task
family.add_task(task)        # add to the family
t2 = family.add_task("t2")   # create a task t2 and add to the family

# Create Task in place
defs = Defs(
         Suite("s1",
            Family("f1",
               Task("t1",
                  Trigger("1==1"),
                  Edit(SLEEP="10"))))) # add Trigger and Variables in place
```

```
aliases
Returns a list of aliases
nodes
Returns a list of aliases
```

class ecflow.TaskVec

Bases: `Boost.Python.instance`

Hold a list of task nodes

append((TaskVec)arg1, (object)arg2) None

extend((TaskVec)arg1, (object)arg2) None

class ecflow.Time

Bases: `Boost.Python.instance`

Is used to define a time dependency

This can then control job submission. There can be multiple time dependencies for a node, however overlapping times may cause unexpected results. The time dependency can be made relative to the beginning of the suite or in repeated families relative to the beginning of the repeated family.

Constructor:

```
Time(string)
  string: i.e.. "00:30" || "00:30 20:00 00:30"   Time(hour,minute,relative&lt;optional&gt; = false)
   int hour:               hour in 24 clock
   int minute:             minute &lt;= 59
   bool relative&lt;optional&gt;: default = False, Relative to suite start or repeated node.

Time(single,relative&lt;optional&gt; = false)
   TimeSlot single:         A single time
   bool relative:           Relative to suite start or repeated node. Default is false

Time(start,finish,increment,relative&lt;optional&gt; = false)
   TimeSlot start:          The start time
   TimeSlot finish:         The finish/end time
   TimeSlot increment:      The increment
   bool relative&lt;optional&gt;: default = False, relative to suite start or repeated node

Time(time_series)
   TimeSeries time_series:Similar to constructor above
```

Exceptions:

- raises IndexError when an invalid Time is specified

Usage:

```
time1 = Time( 10,10 )                                                    #  time 10:10
time2 = Time( TimeSlot(10,10), true)                                     #  time +10:10
time2 = Time( TimeSlot(10,10), TimeSlot(20,10),TimeSlot(0,10), false )  #  time 10:10 20:10 00:10

t = Task("t1",
         time1,time2,time3,
         Time("10:30 20:10 00:10")) # Create time in place
```

time_series((Time)arg1) TimeSeries :

Return the Time attributes time series

class ecflow.TimeSeries

Bases: `Boost.Python.instance`

A TimeSeries can hold a single time slot or a series.

Time series can be created relative to the suite start or start of a repeating node. A Time series can be used as argument to the `ecflow.Time`, `ecflow.Today` and `ecflow.Cron` attributes of a node. If a time the job takes to complete is longer than the interval, a 'slot' is missed e.g time 10:00 20:00 01:00, if the 10.00 run takes more than an hour the 11.00 is missed

Constructor:

```
TimeSeries(single,relative_to_suite_start)
   TimeSlot single :  A single point in a 24 clock
   optional bool relative_to_suite_start : depend on suite begin time or
                                           start of repeating node. Default is false

TimeSeries(hour,minute,relative_to_suite_start)
   int hour   :  hour in 24 clock
   int minute :  minute &lt; 59
   bool relative_to_suite_start&lt;optional&gt; : depend on suite begin time or
                                          start of repeating node. Default is false

TimeSeries(start,finish,increment,relative_to_suite_start)
   start TimeSlot :     The start time
   finish TimeSlot :    The finish time, when used in a series. This must greater than the start.
   increment TimeSlot : The increment. This must be less that difference between start and finish
   bool relative_to_suite_start&lt;optional&gt; : The time is relative suite start, or start of
repeating node.
                                           The default is false
```

Exceptions:

- Raises IndexError when an invalid time series is specified

Usage:

```
time_series = TimeSeries(TimeSlot(10,11),False)
```

finish((TimeSeries)arg1) TimeSlot :

returns the finish time if time series specified, else returns a NULL time slot

has_increment((TimeSeries)arg1) bool :

distinguish between a single time slot and a series. returns true for a series

incr((TimeSeries)arg1) TimeSlot :

returns the increment time if time series specified, else returns a NULL time slot

relative((TimeSeries)arg1) bool :

returns a boolean where true means that the time series is relative

start((TimeSeries)arg1) TimeSlot :

returns the start time

class ecflow.TimeSlot

Bases: `Boost.Python.instance`

Represents a time slot.

It is typically used as an argument to a `TimeSeries` or other time dependent attributes of a node.

Constructor:

```
TimeSlot(hour,min)
   int hour:   represent an hour:
   int minute: represents a minute:
```

Usage:

```
ts = TimeSlot(10,11)
```

empty((TimeSlot)arg1) bool

hour((TimeSlot)arg1) int

minute((TimeSlot)arg1) int

class ecflow.Today

Bases: `Boost.Python.instance`

today is a time dependency that does not wrap to tomorrow.

If the suite s begin time is past the time given for the Today, then the node is free to run.

Constructor:

```
Today(hour,minute,relative&lt;optional&gt; = false)
   int hour               : hour in 24 clock
   int minute             : minute &lt;= 59
   bool relative&lt;optional&gt;: Default = false,Relative to suite start or repeated node.

Today(single,relative&lt;optional&gt; = false)
   TimeSlot single        : A single time
   bool relative          : Relative to suite start or repeated node. Default is false

Today(start,finish,increment,relative&lt;optional&gt; = false)
   TimeSlot start         : The start time
   TimeSlot finish        : The finish/end time. This must be greater than the start time.
   TimeSlot increment     : The increment
   bool relative&lt;optional&gt;: Default = false, Relative to suite start or repeated node.

Today(time_series)
   TimeSeries time_series: Similar to constructor above
```

Exceptions:

- raises IndexError when an invalid Today is specified

Usage:

```
today1 = Today( 10,10 )                                                    #  today 10:10
today2 = Today( TimeSlot(10,10) )                                          #  today 10:10
today3 = Today( TimeSlot(10,10), true)                                     #  today +10:10
today4 = Today( TimeSlot(10,10), TimeSlot(20,10),TimeSlot(0,10), false )  #  time 10:10 20:10 00:10
t = Task("t1",
         today1,today2,today3,today4,
         Today("10:30 20:10 00:10")) # Create today in place
```

time_series((Today)arg1) TimeSeries :

Return the Todays time series

class ecflow.Trigger

Bases: `Boost.Python.instance`

Add a trigger or complete expression.

This defines a dependency for a node. There can only be one trigger or complete expression dependency per node. A node with a trigger can only be activated when the trigger has expired. Triggers can reference nodes, events, meters, variables, repeats, limits and late flag A trigger holds a node as long as the expression returns false.

Exception:

- Will throw RuntimeError if first expression is added as 'AND' or 'OR' expression Like wise second and subsequent expression must have 'AND' or 'OR' booleans set

Usage:

Multiple trigger will automatically be *anded* together, If *or* is required please use bool 'False' as the last argument i.e..

```
task1.add( Trigger("t2 == active" ),
           Trigger("t1 == complete or t4 == complete" ),
           Trigger("t5 == active",False))
```

The trigger for task1 is equivalent to

```
t2 == active and t1 == complete or t4 == complete or t5 == active
```

Since a large number of triggers are of the form *<node> == complete* there are are short cuts, these involves a use of a list

```
task1.add( Trigger( ["t2","t3"] )) #  This is same as t2 == complete and t3 == complete
```

You can also use a node

```
task1.add( Trigger( ["t2",taskx] ))
```

If the node 'taskx' has a parent, we use the full hierarchy, hence we will get a trigger of the form

```
t2 ==complete and /suite/family/taskx == complete
```

If however node taskx has not yet been added to its parent, we use a relative name in the trigger

```
t2 ==complete and taskx == complete
```

get_expression((Trigger)arg1) str :

returns the trigger expression as a string

class ecflow.UrlCmd

Bases: `Boost.Python.instance`

Executes a command ECF_URL_CMD to display a url.

It needs the definition structure and the path to node.

Constructor:

```
UrlCmd(defs, node_path)
   defs_ptr  defs   : pointer to a definition structure
   string node_path : The node path.
```

Exceptions

- raises RuntimeError if the definition is empty
- raises RuntimeError if the node path is empty
- raises RuntimeError if the node path can not be found in the definition
- raises RuntimeError if ECF_URL_CMD not defined or if variable substitution fails

Usage: Lets assume that the server has the following definition:

```
suite s
   edit ECF_URL_CMD  &amp;quot;${BROWSER:=firefox} -remote "openURL(%ECF_URL_BASE%/%ECF_URL%)"&amp;quot;
   edit ECF_URL_BASE &amp;quot;http://www.ecmwf.int&amp;quot;
   family f
      task t1
         edit ECF_URL &amp;quot;publications/manuals/ecflow&amp;quot;
      task t2
         edit ECF_URL index.html
```

```
try:
   ci = Client()
   ci.sync_local()
   url = UrlCmd(ci.get_defs(),"/suite/family/task")
   print(url.execute())
except RuntimeError, e:
    print(str(e))
```

execute((UrlCmd)arg1) None :

Displays url in the chosen browser

class ecflow.Variable

Bases: `Boost.Python.instance`

Defines a variable on a node for use in ecf script.

A Node can have a number of variables. These variables can be added at any node level: Defs, suite, family or task. The variables are names inside a pair of '%' characters in an ecf script. The content of a variable replaces the variable name in the ecf script at job submission time. When a variable is needed at submission time, it is first sought in the task itself. If it is not found, it is sought from the tasks parent and so on, up through the node levels until found. See variable inheritance A undefined variable in a ecf script, causes the task to be aborted, without the job being submitted.

Constructor:

```
Variable(name,value)
   string name: the name of the variable
   string value: The value of the variable

Edit(dict,kwarg) # alternative that allows multiple variables
```

Usage:

```
...
var = Variable("ECF_JOB_CMD","/bin/sh %ECF_JOB% &amp;amp;")
task.add_variable(var)
task.add_variable("JOE","90")
```

The following use example of using Edit, which allow multiple variables to added at the same time

```
t = Task("t1",
          Edit({ "a":"y", "b":"bb"}, c="v",d="b"),
          Edit({ "e":1100, "f":"bb"}),
          Edit(g="d"),
          Edit(h="1"))
```

```
defs = Defs(
          Suite("s1"),
          Edit(SLEEP="1")) # Add user variable to definition
defs.s1 += [ Task("a") ]
defs.s1.a += [ Edit({ "x1":"y", "aa1":"bb"}, a="v",b="b"),
               Edit({ "var":10, "aa":"bb"}),
               Edit(d="d") ]
```

empty((Variable)arg1) bool :

Return true if the variable is empty. Used when returning a Null variable, from a find

name((Variable)arg1) str :

Return the variable name as string

value((Variable)arg1) str :

Return the variable value as a string

class ecflow.VariableList

Bases: `Boost.Python.instance`

Hold a list of Variables

append((VariableList)arg1, (object)arg2) None

extend((VariableList)arg1, (object)arg2) None

class ecflow.Verify

Bases: `Boost.Python.instance`

class ecflow.WhyCmd

Bases: `Boost.Python.instance`

The why command reports, the reason why a node is not running.

It needs the definition structure and the path to node

Constructor:

```
WhyCmd(defs, node_path)
   defs_ptr  defs   : pointer to a definition structure
   string node_path : The node path
```

Exceptions:

- raises RuntimeError if the definition is empty
- raises RuntimeError if the node path is empty
- raises RuntimeError if the node path can not be found in the definition

Usage:

```
try:
   ci = Client()
   ci.sync_local()
   ask = WhyCmd(ci.get_defs(),"/suite/family")
   print(ask.why())
except RuntimeError, e:
    print(str(e))
```

why((WhyCmd)arg1) str :

returns a '/n' separated string, with reasons why node is not running

class ecflow.ZombieAttr

Bases: `Boost.Python.instance`

The zombie attribute defines how a zombie should be handled in an automated fashion

Very careful consideration should be taken before this attribute is added as it may hide a genuine problem. It can be added to any node. But is best defined at the suite or family level. If there is no zombie attribute the default behaviour is to block the init,complete,abort child command. and *f ob* the event,label,and meter child command This attribute allows the server to make a automated response. Please see: `ecflow.ZombieType`, `ecflow.ChildCmdType`, `ecflow.ZombieUserActionType`

Constructor:

```
ZombieAttr(ZombieType,ChildCmdTypes, ZombieUserActionType, lifetime)
   ZombieType            : Must be one of ZombieType.ecf, ZombieType.path, ZombieType.user
   ChildCmdType          : A list(ChildCmdType) of Child commands. Can be left empty in
                           which case the action affect all child commands
   ZombieUserActionType  : One of [ fob, fail, block, remove, adopt ]
   int lifetime&lt;optional&gt;: Defines the life time in seconds of the zombie in the server.
                           On expiration, zombie is removed automatically
```

Usage:

```
# Add a zombie attribute so that child commands(i.e. ecflow_client --init)
# will fail the job if it is a zombie process.
s1 = Suite("s1")
child_list = [ ChildCmdType.init, ChildCmdType.complete, ChildCmdType.abort ]
s1.add_zombie( ZombieAttr(ZombieType.ecf, child_list, ZombieUserActionType.fob))

# create the zombie as part of the node constructor
s1 = Suite("s1",
           ZombieAttr(ZombieType.ecf, child_list, ZombieUserActionType.fail))
```

empty((ZombieAttr)arg1) bool :

Return true if the attribute is empty

user_action((ZombieAttr)arg1) ZombieUserActionType :

The automated action to invoke, when zombies arise

zombie_lifetime((ZombieAttr)arg1) int :

Returns the lifetime in seconds of zombie in the server

zombie_type((ZombieAttr)arg1) ZombieType :

Returns the zombie type

```
child_cmds
The list of child commands. If empty action applies to all child cmds
```

class ecflow.ZombieType

Bases: `Boost.Python.enum`

zombie s are running jobs that fail authentication when communicating with the ecflow_server.

See class zombie type and `ecflow.ZombieAttr` for further information.

```
ecf = ecflow.ZombieType.ecf
names = {'ecf': ecflow.ZombieType.ecf, 'user': ecflow.ZombieType.user, 'path': ecflow.ZombieType.path}
path = ecflow.ZombieType.path
user = ecflow.ZombieType.user
values = {0: ecflow.ZombieType.user, 1: ecflow.ZombieType.ecf, 2: ecflow.ZombieType.path}
```

class ecflow.ZombieUserActionType

Bases: `Boost.Python.enum`

ZombieUserActionType is used define an automated response. See class `ZombieAttr`

This can be either on the client side or on the server side

client side:

- fob: The child command always succeeds, i.e. allows job to complete without blocking
- fail: The child command is asked to fail.
- block: The child command is asked to block.
  This is the default action for init,complete and abort child commands

server side:

- adopt: Allows the password supplied with the child command s, to be adopted by the server
- kill: Kills the zombie process associated with the child command using ECF_KILL_CMD.
  path zombies will need to be killed manually. If kill is specified for path zombies they will be fobed, i.e. allowed to complete without blocking the job.
- remove: ecflow_server removes the zombie from the zombie list.
  The child continues blocking. If the process is still running, the zombie may well re-appear

Note: Only adopt will allow the child command to continue and change the node tree

```
adopt = ecflow.ZombieUserActionType.adopt
block = ecflow.ZombieUserActionType.block
fail = ecflow.ZombieUserActionType.fail
fob = ecflow.ZombieUserActionType.fob
kill = ecflow.ZombieUserActionType.kill
names = {'adopt': ecflow.ZombieUserActionType.adopt, 'remove': ecflow.ZombieUserActionType.remove, 'kill':
ecflow.ZombieUserActionType.kill, 'fob': ecflow.ZombieUserActionType.fob, 'fail': ecflow.ZombieUserActionType.
fail, 'block': ecflow.ZombieUserActionType.block}
remove = ecflow.ZombieUserActionType.remove
values = {0: ecflow.ZombieUserActionType.fob, 1: ecflow.ZombieUserActionType.fail, 2: ecflow.
ZombieUserActionType.adopt, 3: ecflow.ZombieUserActionType.remove, 4: ecflow.ZombieUserActionType.block, 5:
ecflow.ZombieUserActionType.kill}
```

`ecflow.debug_build()` bool