spawn Documentation

Release 0.2.1+15.g488a03c.dirty

Michael Tinning, Philip Bradstock

Mar 26, 2020

USER GUIDE

Spawn is a python package that allows users to concisely specify and execute a large number of tasks with complex and co-dependent input parameter variations. It is used particularly in engineering where frequently thousands of similar simulations with input parameter variations are run for design and certification purposes. Spawn allows the specification of such large task sets to be formulated in a concise and readable input file.

A typical Spawn process is: 1. Write an input specification file in JSON that specifies the parameters and their values. 2. Inspect the fully expanded specification tree with the *inspect* command. 3. Execute all tasks with the *run* command. This uses a back-end "spawner" to create the tasks, which can be fully customised for individual use cases.

If you are interested in using Spawn for executing wind turbine aeroelastic simulation using NREL's FAST, please see the spawn-wind page.

Read *Spawn Input File Definition* to discover how to build your first Spawn spec file.

If you're using the command line interface, take a look at *Spawn Command Line Interface*.

A full *API Reference* is also provided.

CHAPTER

ONE

SPAWN INPUT FILE DEFINITION

Spawn input is a hierarchical structure of branching nodes which allows large numbers of parameter sets (referred to as "specification nodes") to be specified in a declarative manner. The input is defined in JSON format (see http://json.org for the JSON standard). JSON editors are widely available, or you can use your favourite text editor to write Spawn input files.

1.1 Getting Started

The specification is defined in an object named `"spec"`. Each name/value pair within this object is a parameter name and its value. The following generates a single specification node with one parameter, named `"alpha"` with a value of 4:

```
{
    "spec": {
        "alpha": 4
    }
}
```

Sibling name/value pairs are simultaneous (i.e. occur on the same node). The following generates a single node with *two* simultaneous parameters - `"alpha"` with a value of 4, and "beta" with a value of "tadpole":

```
{
    "spec": {
        "alpha": 4,
        "beta": "tadpole"
    }
}
```

Separate nodes can be created by separating parameters into different JSON nodes. Parameters defined outside of the object are also applied on each node. The following creates two nodes, both with parameters named `"alpha"` and `"beta"`, where the first node has parameter values of 4 and "tadpole" respectively and the second has values of 6 and "tadpole" respectively. Note that the names of the sub-objects (`"blah"` and `"blo"`) do not contribute to the parameter specification:

```
{
    "spec": {
        "beta": "tadpole",
        "blah": { "alpha": 4},
        "blo": { "alpha": 6}
    }
}
```

An identical specification could be written (less concisely) as:

```
{
    "spec": {
        "blah": { "alpha": 4, "beta": "tadpole"},
        "blo": { "alpha": 6, "beta": "tadpole"}
    }
}
```

Avoiding repetitive definition and enabling concise and readable but complex specifications is one of the key aims of Spawn.

1.2 Arrays

Multiple specification nodes where one parameter is varied can be created by using the array property of JSON. The same specification as at the end of the last section (two nodes, both with parameter "beta" of "tadpole" and parameter "alpha with values of 4 and 6) can be created by:

```
{
    "spec": {
        "alpha": [4, 6],
        "beta": "tadpole"
    }
}
```

1.2.1 Cartesian Product

The automatic behaviour of multiple sibling arrays is to create all the parameter combinations of them (i.e. apply Cartesian Product). The following will create 6 (3*2) nodes (2D product):

```
{
    "spec": {
        "alpha": [3, 5, 8],
        "beta": ["tadpole", "frog"]
    }
}
```

Additional sibling arrays will add additional dimensions to the Cartesian product, and there is no limit. In this manner, a very large number of nodes can be created with just a few lines.

1.2.2 Zip

To apply a one-one mapping between, we apply the zip "combinator" on the two arrays. A "combinator" is a name/object pair where the name determines the combination to be performed. The name starts with a # to differentiate it from other name/object pairs. The following generates three nodes ((3, "egg"), (5, "tadpole"), (8, "frog")):

```
{
    "spec": {
        "combine:zip": {
            "alpha": [3, 5, 8],
            "beta": ["egg", "tadpole", "frog"]
        }
    }
}
```

There is no limit to the number of sibling arrays, but they *must* all have equal size.

1.3 Value Proxies

The value of parameter/value pairs can be represented by a proxy. The proxy is a string that starts with either a type identifier followed by a colon (longhand) or a special character (shorthand) to determine which type of value proxy it is. The parser then replaces the proxy when the specification is resolved. The types of value proxies are as follows:

Type	Long-hand	Short-hand	Description
Macro	`macro:`	`$`	Direct substitution of a previously declared value
Generator	`gen:`	`@`	Generates a (in general different) value each time it is resolved
Evaluator	`eval:`	`#`	Evaluates a value based on a deterministic function which can take arguments

1.3.1 Macros

Macros are declared alongside the spec and can then be used repeatedly. The name of the name/value pairs in the `macros` object determines the name of the macro that can be used in the `spec` object (where it must be prefixed). They can be a single value, array or object. The following will produce six nodes (three values of `"alpha"` with `"beta"` parameter specified, and three more with `"gamma"` parameter specified):

```
{
    "macros": {
        "Alphas": [3, 5, 8]
    },
    "spec": {
        "a": {
            "alpha": "macro:Alphas",
            "beta": "tadpole"
        },
        "b": {
            "alpha": "$Alphas",
            "gamma": 4.2
        }
    }
}
```

1.3.2 Generators

Generators generate a value each time they are resolved in the specification. The main uses of generators is providing random variates and counters. Generators are declared in an object named `generators`. Each generator is a name/object pair, where the name specifies the name of the generator to be used in the `spec` object. Each generator object must specify its method and arguments to its constructor. These are the inbuilt constructors:

method	Argument names (default value)	Description
IncrementalInt	start(1), step(1)	An incrementing integer starting at start and incrementing by step each time it is resolved
RandomInt	min(1), max(999), seed(1)	A random integer between min and max each time it's resolved

The following example generates a value of 4 for "alpha" via the "a" object and a value of 5 via the "b" node:

```
{
    "generators": {
        "Counter": {
            "method": "IncrementalInt",
            "start": 4
        }
    },
    "spec": {
        "a": {
            "alpha": "@Counter",
            "beta": "tadpole"
        },
        "b": {
            "alpha": "gen:Counter",
            "gamma": 4.2
        }
    }
}
```

1.3.3 Evaluators

Evaluators allow function-style syntax to evaluate expressions with arguments. Arithmetic operations are supported as well as inbuilt evaluators range, which produces an evenly spaced array, and repeat, which repeats a particular value. Unlike macros and generators, evaluators do not need an object defined alongside the spec. Some examples:

Example	Resolution
"#3 + 5"	8
"#3 - 5"	-2
"#3 * 5"	15
"#3 / 5"	0.6
"#range(3, 8)"	[3, 4, 5, 6, 7, 8]
"#range(0.3, 0.5, 0.1)"	[0.3, 0.4, 0.5]
"eval:repeat(5, 3)"	[5, 5, 5]

Note that the repeat can be used with a generator as argument and therefore generate a different value for each element of the array. Evaluators can also take other parameters simultaneously present in the specification if they are prefixed by !. They do not need to be in the same object, but if not they must be defined higher up the object tree (i.e. they are not referenceable if in sub-objects). The following resolves "gamma" into the list [3, 4]:

```
{
    "spec": {
        "alpha": 3,
        "blah": {
```

(continues on next page)

(continued from previous page)

```
            "beta": 5,
            "gamma": "#range(!alpha, !beta)"
        }
    }
}
```

When referencing a parameter in an arithmetic operation, the `#` is no longer needed (but the `!` is required):

```
{
    "spec": {
        "alpha": 4,
        "beta": "!alpha + 3"
    }
}
```

1.4 Literals

There are cases in which it is desired that specification value does not take on its default spawn interpretation. For this, there is the concept of a literal. This is done by prefixing the parameter name with `~`. The following will generate a single node where `alpha` is an array (passed through to the spawner as a list) and `beta` is a string starting with `$` (rather than looking up a macro). Arrays, objects, all (apparent) value proxies and equations can be taken as literal and therefore not expanded or looked up:

```
{
    "spec": {
        "~alpha": ["egg", "tadpole", "frog"],
        "~beta": "$NotAMacro"
    }
}
```

Literals can also be specified on the value-side. This can be particularly useful when it is desired to expand literals as part of an expansion. In this case, the value must always be a string, but if the string succeeding the literal prefix is JSON serialisable it will be serialised as such, otherwise the value will remain a string. For example, the following produces three nodes, each with an array as the value of the `alpha` parameter:

```
{
    "spec": {
        "alpha": ["~[1, 2]", "~[3, 4]", "~[5, 6, 7]"]
    }
}
```

1.5 Policies

Policies do not generate parameters but provide additional information for the spawner to work with. The use of the end policy is determined by the spawner. The only policy at current is the path policy

1.5.1 Path

This may generally be interpreted as a file path but could also be for example a URL endpoint or any other kind of path. All specification nodes have a path associated with them, whether specified by the user or not. In the case of simulations, this path can be used to determine where the output of the simulation is saved. The following produces one specification node with the path `my_path`

```
{
    "spec": {
        "policy:path": "my_path",
        "alpha": "tadpole"
    }
}
```

Paths that are declared at different levels of the tree are appended as sub-folders. For example the following will produce a single node with the path `my/path`:

```
{
    "spec": {
        "policy:path": "my",
        "alpha": "tadpole",
        "blah": {
            "policy:path": "path",
            "beta": 2
        }
    }
}
```

Distinct nodes in the tree *always* have different paths. If there are two nodes that resolve to the same path, then they will be put into lettered sub-folders `a`, `b`, `c` etc... For example, the following produces the paths `my_path/a`, `my_path/b`, `my_path/c`:

```
{
    "spec": {
        "policy:path": "my_path",
        "alpha": ["egg", "tadpole", "frog"]
    }
}
```

In order to make more meaningful paths, parameter values (of any type that is convertible to string) can also be inserted in the path, by using `{name}` syntax in the path. For example, the following produces the paths `a_egg`, `a_tadpole`, `a_frog`:

```
{
    "spec": {
        "policy:path": "a_{alpha}",
        "alpha": ["egg", "tadpole", "frog"]
    }
}
```

Sub-folders can also be assigned in a single `policy:path` value by adding `/` in the path:

```
{
    "spec": {
        "policy:path": "{alpha}/{beta}",
        "alpha": ["egg", "tadpole", "frog"],
        "beta": [1, 2, 3]
```

(continues on next page)

(continued from previous page)

```
    }
}
```

Rather than using the value of a parameter in the path, an incremental digit or letter can be used instead by introducing the syntax `{name:1}` or `{name:a}` respectively. For example, the following produces the paths `alpha_1`, `alpha_2`, `alpha_3`:

```
{
    "spec": {
        "policy:path": "alpha_{alpha:1}",
        "alpha": ["egg", "tadpole", "frog"]
    }
}
```

Sequences can also be made to start at other letters/digits by using the syntax `{name:4}` (which would start at 4) for example. Here is a full table the alphanumeric indicators that can be used:

Identifier	Sequence
`a`	a, b, c, ... aa, ab, ...
`f`	f, g, h, ... aa, ab, ...
`1`	1, 2, 3, ... 10, 11, ...
`5`	5, 6, 7, ... 10, 11, ...
`aa`	aa, ab, ... ba, bb, ...
`01`	01, 02, ... 10, 11, ...

1.6 Indexing

Spawners may define certain parameters as arrays. In order to set the values of elements in an array parameter, the specification allows indexing with use of square brackets suffixing the name parameter pair. For example, `"alpha[1]": 3` sets the first element of the `"alpha"` parameter array to 3. Whether the index is 0-based or 1-based is ultimately the decision of the spawner (which gets passed the index) but by convention is generally 1-based.

CHAPTER

TWO

SPAWN COMMAND LINE INTERFACE

Command Line Interface for spawn

2.1 spawn

Command Line Interface

```
spawn [OPTIONS] COMMAND [ARGS]...
```

Options

--log-level <log_level>
The log level

Options error|warning|info|debug

--log-console
Write logs to the console

-d <d>
Definitions to override configuration file parameters (e.g. -d spawn.workers=2)

--config-file <config_file>
Path to the config file.

2.1.1 check-config

Check the configuration. Parses the current configuration and prints to stdout

```
spawn check-config [OPTIONS]
```

2.1.2 inspect

Expand and write to console the contents of the SPECFILE

```
spawn inspect [OPTIONS] SPECFILE
```

Options

-o, --outfile <outfile>
write inspection output to file rather than to console

-f, --format <format>
format of specification inspection

Options txt|json

Arguments

SPECFILE
Required argument

2.1.3 run

Runs the SPECFILE contents and write output to OUTDIR

```
spawn run [OPTIONS] SPECFILE OUTDIR
```

Options

--type <type>
The type of runs to create. Must have a corresponding plugin.

--local, --remote
Run local or remote. Remote running requires a luigi server to be running

Arguments

SPECFILE
Required argument

OUTDIR
Required argument

2.1.4 stats

Analyse the SPECFILE and print stats to the screen

```
spawn stats [OPTIONS] SPECFILE
```

Arguments

SPECFILE
Required argument

2.1.5 work

Adds a worker to a remote scheduler

```
spawn work [OPTIONS]
```

CHAPTER

THREE

GLOSSARY

Term	Definition
Combinator	Functor that combines multiple parameter arrays e.g. Cartesian outer product and "zip". They are defined in the key string with the prefix `combine:`
Evaluator	Object that generates a value determinsiticly based on input arguments. In the input it is referenced by using a string value that is prefixed with `eval:` or `#`
Generator	Object that generates a (different) value each time it is referenced. In the input it is declared in a `generators` top-level section and referenced using a string value that is prefixed with `gen:` or `@`
Macro	Direct substitution of a value with another value defined in the `macro` top-level section. Referenced using a string value that is prefixed with `macro:` or `$`
Spawner	The object that is responsible for creating ("spawning") simulation or calculations based on a specification tree that is parsed from the input
Specification Node	A single node in the specification tree with any given number of parameters associated with it. Generally the spawner will interpret spawn one simulation for each specification node.

CHAPTER

FOUR

API REFERENCE

spawn.cli	Command Line Interface for spawn
spawn.parsers	Parsers for spawn spec files
spawn.runners	Runners for `spawn` tasks
spawn.schedulers	Schedulers for `spawn`
spawn.simulation_inputs	Simulation inputs
spawn.spawners	Task Spawners for `spawn`
spawn.specification	Specification definition
spawn.tasks	Simulation Tasks
spawn.util	Utility classes and functions used by `spawn`

CHAPTER

FIVE

SPAWN.PARSERS

Parsers for spawn spec files

This module contains any classes, functions and utilities related to parsing data

class spawn.parsers.**DictSpecificationProvider**(*spec*)
Implementation of *SpecificationDescriptionProvider* that reads the specification from a provided dict

get()
Gets the specification

Returns A dict representation of the description

Return type dict

class spawn.parsers.**SpecificationDescriptionProvider**
Abstract base class for implementations that provide the specification description.

get()
Gets the specification description

Returns A dict representation of the description

Return type dict

class spawn.parsers.**SpecificationFileReader**(*input_file*)
Implementation of *SpecificationDescriptionProvider* that reads the specification from a file

get()
Reads the specification description from a file

Returns A dict representation of the description

Return type dict

class spawn.parsers.**SpecificationNodeParser**(*value_proxy_parser*, *combinators=None*, *default_combinator=None*)
Expands the specification nodes, starting at the `node_spec` provided

Given a starting *node_spec*, the specification `node_spec` parser assesses child nodes and expands them according to their values.

parse(*node_spec*, *parent=None*, *node_policies=None*, *ghost_parameters=None*)
Parse the *node_spec*, and expand its children.

This iterates through a *node_spec* and expands it's children.

Parameters

- **node_spec** (*dict*) – A node specification

- **parent** (SpecificationNode) – A specification node to add the new nodes to

Returns The expanded *node_spec*

Return type SpecificationNode

class spawn.parsers.**SpecificationParser**(*plugin_loader*)

Class for parsing specifications

Given a specification provider, the *SpecificationParser* will get the specification and produce a tree representation of the nodes.

parse(*description*)

Parse the specification description

Reads the metadata from the file and creates any required value libraries, before initialising a *SpecificationNodeParser* to expand the nodes defined in the description.

Parameters **description** (*dict*) – The specification description

Returns An object representing the expanded specification tree.

Return type SpecificationModel

class spawn.parsers.**ValueLibraries**(*generators=None*, *macros=None*, *evaluators=None*)

Container class for value libraries (generators, macros & evaluators)

copy()

Copies the values library

Returns A copy of this object

Return type *ValueLibraries*

property evaluators

Evaluators library

property generators

Generators library

property macros

Macros library

CHAPTER

SIX

SPAWN.RUNNERS

Runners for `spawn` tasks

A runner has a `run` method, which runs the work required for a task, and a `complete` method, which returns `True` when the task has completed.

class spawn.runners.**ProcessRunner**(*id_*, *input_file_path*, *exe_path*, *run_name=None*, *output_dir=None*, *cwd=None*)

Runner that uses the native os process (provided by `subprocess`) in order to run tasks

complete()

Determine if the run is complete.

Returns `True` if the run is complete; `False` otherwise.

Return type bool

error_logs()

Error logs produced by the process, if any

Returns The output written to stderr, if any, otherwise `None`

Return type str

logs()

Stdout logs produced by the process, if any

Returns The output written to stdout, if any, otherwise `None`

Return type str

property output_file_base

The base path of the output file (without extension)

Returns The path to the output file, without extension.

Return type path-like

property process_args

The arguments to be provided to the subprocess.

Can be overridden in derived classes.

Returns An array containing the process arguments

Return type list

run()

Runs the process synchronously.

Runs the process synchronously and when complete writes the log files and a status file.

property state_file
The path to the state file

Returns The path to the state file

Return type path-like

CHAPTER

SEVEN

SPAWN.SCHEDULERS

Schedulers for `spawn`

A scheduler understands how to turn a spec into a list of jobs and related dependncies to run

class spawn.schedulers.**LuigiScheduler**(*config*)

Scheduler implementation for Luigi

Because this is currently the only scheduler implementation it's probable that the interface will evolve in time.

add_worker()

Add a worker

run(*spawner*, *spec*)

Run the spec by generating tasks using the spawner

Parameters

- **spawner** (`TaskSpawner`) – The task spawner
- **spec** (`SpecificationModel`) – The specification

CHAPTER

EIGHT

SPAWN.SIMULATION_INPUTS

Simulation inputs

An input a class that reads and writes parameter values, normally to a file

class spawn.simulation_inputs.**JsonSimulationInput**(*parameter_set*, ***write_options*)
A dictionary input written as a JSON file where the parameter set is deep copied

class spawn.simulation_inputs.**SimulationInput**
Handler of inputs for a simulation that will typically be parsed from and written to a file

classmethod from_file(*file_path*)
Creates a *SimulationInput* by loading a file

Parameters **file_path** (*path-like*) – The file path to load

Returns The simulation input object

Return type An instance of *SimulationInput*

abstract hash()
Returns a hash of the contents of the file

Returns The hash

Return type str

abstract to_file(*file_path*)
Writes the contents of the input file to disk

Parameters **file_path** (*path-like*) – The path of the file to write

CHAPTER

NINE

SPAWN.SPAWNERS

Task Spawners for `spawn`

Task spawners turn specification nodes into tasks that can be submitted by a scheduler

class spawn.spawners.**SingleInputFileSpawner**(*simulation_input*, *file_name*)
Runs bespoke executable taking a single input file as its only command line argument

branch()
Deep copy task input and dependencies so that they can be edited without affecting trunk object

spawn(*path_*, *metadata*)
Create new derivative of luigi.Task for later execution

class spawn.spawners.**TaskSpawner**
Base class task spawner

branch()
Deep copy task input and dependencies so that they can be edited without affecting trunk object

spawn(*path_*, *metadata*)
Create new derivative of luigi.Task for later execution

CHAPTER

TEN

SPAWN.SPECIFICATION

Specification definition

The *SpecificationModel* contains the definition of the tasks to be spawned

class spawn.specification.**DictSpecificationConverter**
Class for converting specification models

Converts *SpecificationModel* into dict

convert(*spec*)
Converts the given spec model into a dict

Parameters **spec** (*SpecificationModel*) – The specification model to convert

Returns A dict represenation of the specification model

Return type dict

class spawn.specification.**Evaluator**(**args*, *name=None*)
Evaluator base class implementation of *ValueProxy*

Implements the *evaluate()* method of the parent class to expand any arguments

evaluate(***kwargs*)
Evaluates the value proxy.

Expands any arguments that are evaluators, and calls the _evaluate() implementation required by base class

class spawn.specification.**Macro**(*value*)
Implementation of *ValueProxy* that can contain a value

evaluate()
Evaluates the *Macro* - returns the value

Returns The value contained within the *Macro*

Return type object

class spawn.specification.**SpecificationMetadata**(*spec_type*, *creation_time*, *notes*)
Container class for the *SpecificationModel* metadata

property creation_time
The creation time

property notes
Notes related to the specification model

property spec_type
The type of this specification

class spawn.specification.**SpecificationModel**(*base_file*, *root_node*, *metadata*)
Class to contain the description of the spawn specification

property base_file
The base file

property metadata
The metadata

property root_node
The root node

class spawn.specification.**SpecificationNode**(*parent*, *property_name*, *property_value*, *path*, *ghosts*)
Tree node representation of the nodes of the specification

add_child(*child*)
Adds a child to this node

Parameters **child** (*SpecificationNode*) – The child node to add

property children
Get the children of this node

Returns Child nodes

Return type list

property collected_indices
Gets the property names and indicies of this node and all ancestor nodes

Returns A dict containing the properties of this node and all ancestor nodes

Return type dict

property collected_properties
Gets the properties and values of this node and all ancestor nodes

Returns A dict containing the properties of this node and all ancestor nodes

Return type dict

copy(*new_parent*)
Copies this node and this node's children

Parameters **new_parent** (*SpecificationNode*) – The new parent node

Returns A copy of this node

Return type *SpecificationNode*

classmethod create_root(*path=None*)
Create a root node

Parameters **path** (*str*) – The path for the root node

Returns A root specification node (without parents)

Return type *SpecificationNode*

property description
Description of this node

Returns A description of the node

Return type str

evaluate()
Evaluates all children in this node

property ghosts
Returns the collected ghost parameters

Returns The ghost parameters for this node

Return type dict

property has_property
Does this node have a property value

Returns `True` if this node has a type that contains properties

Return type bool

property index
Gets the index of this node in the parent's child nodes

Returns The index if this node is not a root node; otherwise -1

Return type int

property is_root
Is this the root node

Returns `True` if this node is the root; otherwise `False`

Return type bool

property leaves
Gets the leaf nodes descended from this node

Returns The leaf nodes

Return type list

property parent
Get the parent

Returns The parent node

Return type *SpecificationNode*

property path
The path for this node.

Used as a key to locate the ouputs. Evaluate using the path property and the collected properties and indices at this node.

Returns The path for this node

Return type str

property property_name
Gets the property name for this node

Returns The property name

Return type str

property property_value
Gets the property value for this node

Returns The property value

Return type object

property root
Gets the root node from this node

Returns The root node

Return type *SpecificationNode*

class spawn.specification.**SpecificationNodeFactory**
Factory class for creating *SpecificationNode* objects

create(*parent*, *name*, *value*, *path*, *ghosts*, *children=None*, *literal=False*)
Creates a *SpecificationNode*, based on the value

Parameters

- **parent** (*SpecificationNode*) – The parent *SpecificationNode*
- **name** (*str*) – The name of the node
- **value** (*object*) – The value of the node
- **ghosts** (*dict*) – Ghost values
- **children** (*list*) – The children of the new node, if any
- **literal** (*bool*) – if True, the value is not expandable and is set literally

class spawn.specification.**ValueProxy**
Base value proxy class

A value proxy is anything that can be *evaluate* d in place of a value

evaluate()
Evaluates the *ValueProxy*

Must be implemented in a derived class

Returns A value

Return type object

class spawn.specification.**ValueProxyNode**(*parent*, *name*, *value_proxy*, *path*, *ghosts*)
Implementation of *SpecificationNode* that allows a *ValueProxy* definition of a node

evaluate()
Evaluates this node to determine what it's value should be.

Replaces children with new values generated by this node. Subsequently evaluates all children.

spawn.specification.**evaluate**(*value_proxy*, **args*, ***kwargs*)
Utility function to evaluate a *ValueProxy*

Determines whether *kwargs* needs to be provided

Parameters **value_proxy** (VaWlueProxy) – The *ValueProxy* to evaluate

CHAPTER

ELEVEN

SPAWN.TASKS

Simulation Tasks

class spawn.tasks.**SimulationTask**(**args*, ***kwargs*)
Implementation of `luigi.Task`

property available_runners
Runners available for this task.

Can be overridden by derived tasks

complete()
Determine if this task is complete

Returns `True` if this task is complete; otherwise `False`

Return type bool

on_failure(*exception*)
Interprets any exceptions raised by the run method.

Attempts to find any logs associated with the runner.

Returns A string representation of the error.

Return type str

run()
Run this task

property run_name_with_path
Return the run name of this task

class spawn.tasks.**SpawnTask**(**args*, ***kwargs*)
Implementation of `luigi.Task` that defines ID and dependencies parameters

complete()
Determine if this task is complete

Returns `True` if this task is complete; otherwise `False`

Return type bool

property metadata
Metadata for this task

requires()
The prerequisites for this task

run()
Run the task. Derived classes should implement this method.

class spawn.tasks.**TaskListParameter**(*default=<object object>*, *is_global=False*, *significant=True*, *description=None*, *config_path=None*, *positional=True*, *always_in_help=False*, *batch_method=None*, *visibility=<ParameterVisibility.PUBLIC: 0>*)

Implementation of `luigi.Parameter` to allow definitions of multiple tasks as dependencies

parse(*x*)

Parse the string

serialize(*x*)

Serialize this object

CHAPTER

TWELVE

SPAWN.UTIL

Utility classes and functions used by `spawn`

class spawn.util.**ArrayProperty**(*type_*, *fget=None*, *fset=None*, *fdel=None*, *fvalidate=None*, *default=None*, *doc=None*, *abstract=False*, *readonly=False*)

Implementation of `PropertyBase` for array properties `__get__()`, `__set__()` and `__delete__()` return array wrappers that allow indexes to be used

class spawn.util.**FloatProperty**(*fget=None*, *fset=None*, *fdel=None*, *fvalidate=None*, *default=None*, *doc=None*, *abstract=False*, *readonly=False*, *min=None*, *max=None*)

Implementation of `NumericProperty` for float properties

class spawn.util.**IntProperty**(*fget=None*, *fset=None*, *fdel=None*, *fvalidate=None*, *default=None*, *doc=None*, *abstract=False*, *readonly=False*, *min=None*, *max=None*)

Implementation of `NumericProperty` for int properties

class spawn.util.**PathBuilder**(*path=''*)

The path builder class

This class provides a standardised way to build paths and interpolate/format them with values

format(*properties*, *indices=None*)

Format the path by interpolating with the properties provided.

If indices are provided, use them in place of the properties.

Parameters

- **properties** (*dict*) – The properties to interpolate with.
- **indices** – The indicies to use in place of the properties, if any. Defaults to `None`.

Returns A new instance with the interpolated values

Return type *PathBuilder*

static index(*index*, *index_format='1'*)

Formats an index given the `index_format` provided

Parameters

- **index** (*int*) – The index to format
- **index_format** (*str*) – The index format (see below for examples).

Returns The formatted index

Return type str

Format	Description	Examples
0*[d]+	Padded integer with start value	01 -> 01, 02, 03... 002 -> 002, 003, 004...
[a]+	Alphabetical	a -> a, b, c, d... aa -> aa, ab, ac, ad...

join(*other*)
Join two paths by adding `other` onto the end of `self`

Parameters **other** (*str*) – The path to add to the end of this path

Returns A new path builder with the joined paths

Return type *PathBuilder*

join_start(*other*)
Join two paths by adding `other` onto the start of `self`

Parameters **other** (*str*) – The path to add to the start of this path

Returns A new path builder with the joined paths

Return type *PathBuilder*

class spawn.util.**StringProperty**(*fget=None*, *fset=None*, *fdel=None*, *fvalidate=None*, *default=None*, *doc=None*, *abstract=False*, *readonly=False*, *possible_values=None*, *regex=None*)
Implementation of *TypedProperty* for string properties

class spawn.util.**TypedProperty**(*type_*, *fget=None*, *fset=None*, *fdel=None*, *fvalidate=None*, *default=None*, *doc=None*, *abstract=False*, *readonly=False*)
Base class for typed properties

spawn.util.**configure_logging**(*log_level*, *command_name*, *log_console=True*, *log_file=True*)
Configure logging

Parameters

- **log_level** (*str*) – The log level (error, warning, info, debug, trace)
- **command_name** (*str*) – The name of the invoked subcommand
- **log_console** (*bool*) – `True` if logs should be displayed in the console; otherwise `False`. Defaults to `True`.
- **log_file** (*bool*) – `True` if logs should be written to file; otherwise `False`. Defaults to `True`.

spawn.util.**float_property**(*fget*)
Function decorator for *FloatProperty*

spawn.util.**int_property**(*fget*)
Function decorator for *IntProperty*

spawn.util.**prettyspec**(*spec*, *outstream=None*)
Writes the given spec tree to the stream provided

spawn.util.**quote**(*strpath*)
Wrap the given string in quotes

Parameters **strpath** (*str*) – A string representing a path

spawn.util.**string_property**(*fget*)
Function decorator for *StringProperty*

spawn.util.**typed_property**(*type_*)
Function decorator for *TypedProperty*

CHAPTER

THIRTEEN

INDICES AND TABLES

- genindex
- modindex
- search

PYTHON MODULE INDEX

S

INDEX

Symbols

A

B

C

D

E

S

T

V