Morpher

API Documentation

November 17, 2011

Contents

1 Package morpher.misc

Contains various modules that are shared by more than one package or do not fall neatly into the scope of other packages.

Currently contains `config`, a class used to share configuration information between all components of a project; `log_setup`, which contains a simple method that initializes the project-wide logging system; `status_reporter`, which contains a class for tracking and displaying progress in the form of a status bar; and `section_parameter`, which builds off of `status_reporter` to report the progress of a multi-part program.

(GRAPH)

Author: Rob Waaser

Contact: robwaaser@gmail.com

Organization: Carnegie Mellon University

Since: October 22, 2011

1.1 Modules

- **config**: Contains the `Config` class definition
  (Section 2, p. 3)
- **log_setup**: Contains the `setupLogging` and `translateLevel` function definitions, used for interacting with the standard Python `logging` module
  (Section 3, p. 6)
- **section_reporter**: Contains the `SectionReporter` class definition for reporting progress updates
  (Section 4, p. 9)
- **status_reporter**: Contains the `StatusReporter` class definition for reporting progress updates
  (Section 5, p. 12)

2 Module morpher.misc.config

Author: Rob Waaser

Contact: robwaaser@gmail.com

Organization: Carnegie Mellon University

Since: October 22, 2011

2.1 Variables

Name	Description
__package__	**Value:** `'morpher.misc'`

2.2 Class Config

ConfigParser.RawConfigParser
ConfigParser.ConfigParser
morpher.misc.config.Config

A wrapper for Python's `ConfigParser` class which adds some project-specific configuration and a toString method.

Inherits from Pythons' standard `ConfigParser` class, which is used to read in files in the well-known INI format and parse them for configuration information. Config overrides the `__init__` method with it's own version, which does some project-specific configuration, and also adds a toString method, which returns a pretty-printed string useful for logging the state of this Config object.

Config is designed to be used as a central registry of configuration information for a project, and after it is initialized with the contents of a configuration file, it should be passed to every object in the project that needs to access configuration information. Each object can then use their individual reference to the global Config object to read and write key-value pairs as necessary, which can be seen by all other objects as well.

Note: Config is naturally pickleable as long as no key-value pairs are added that contain pickleable objects - meaning it can be used to store a programs' state to a file and used to later restore that state.

To Do: Add additional validation of parameters read from the config file

2.2.1 Methods

__init__(*self*, ***params*)

Parses a configuration file and any additional keyword parameters to create and initialize a new configuration object.

The __init__ method accepts a list of optional keyword arguments, reads in additional arguments from a configuration file, and also contains a list of default parameter values. The final value of a particular parameter is set to (in order of precedence):

1. The supplied keyword parameter, if one is given
2. The supplied value in the configuration file, if one is given
3. The built-in default value (if one exists for this parameter)

The initialization process does not use the logging system like the rest of Morpher, since the logging system is dependent on configuration information supplied here.

Refer to the documentation for `ConfigParser` for information on how config files are parsed and how key-value pairs can be read and written.

Parameters

`params:`	Override values for optional keyword arguments *(type=keyword options)*
`configfile:`	The path to the configuration file
`debug:`	A boolean value enabling debug mode if *True*
`dll:`	The path to the target dll (no default)
`listfile:`	The path to the collection listfile (no default)

Raises

`Exception` An exception is raised if a needed parameter is not found in the params, config file, or default values.

Overrides: ConfigParser.RawConfigParser.__init__

Note: Config defines the default option "basedir" as the path to the current working directory. Entries in the config file can use this option to refer to other directories relative to the current directory, for example: %(BASEDIR)s\data

toString(*self*)

Returns a pretty-printed string suitable for displaying or logging the contents of this Config object

Returns a string similar to the following:

```
Configuration dump:
[TEMP]
  basedir : C:\Users\Rob\workspace\ApiFuzzing
[directories]
  basedir : C:\Users\Rob\workspace\ApiFuzzing
  data : C:\Users\Rob\workspace\ApiFuzzing\data
  tools : C:\Users\Rob\workspace\ApiFuzzing ools
  logs : C:\Users\Rob\workspace\ApiFuzzing\logs
[logging]
  basedir : C:\Users\Rob\workspace\ApiFuzzing
  enabled : yes
  level : debug
```

Return Value

Nicely-formatted string containing contents of the Config object

(type=string)

Inherited from ConfigParser.ConfigParser

get(), items()

Inherited from ConfigParser.RawConfigParser

add_section(), defaults(), getboolean(), getfloat(), getint(), has_option(), has_section(), options(), optionxform(), read(), readfp(), remove_option(), remove_section(), sections(), set(), write()

2.2.2 Class Variables

Name	Description
Inherited from ConfigParser.RawConfigParser	
OPTCRE, OPTCRE_NV, SECTCRE	

3 Module morpher.misc.log_setup

Contains the `setupLogging` and `translateLevel` function definitions, used for interacting with the standard Python `logging` module

Author: Rob Waaser

Contact: robwaaser@gmail.com

Organization: Carnegie Mellon University

Since: November 1, 2011

3.1 Functions

setupLogging(*cfg*, *root*=`None`)

When called with a Config object, uses the Config object to extract configuration information and sets up Python's standard `logging` system for project-wide use.

Initializes the log system provided by Python's `logging` module using information in a provided `Config` object. The log system defines the top-level package in the heirarchy of this module as the root logger by default, or a supplied root can be used instead.

The following actions are performed:

- The logging->enabled option in the cfg object is checked and used to either enable or disable logging globally
- Sets up a handler that prints logging messages of level logging.ERROR or higher to standard output
- Sets up a handler that prints all other logging messages to a log file, located in the directory specified in directories->logging
- Stops propagation of messages above the defined root logger
- Registers an `atexit` handler that ensures the logging system is properly flushed upon program exit.

Parameters

`cfg:` A `Config` object containing logging setup information

(type=`Config` object)

`root:` An optional string specifying the name of the root module

(type=string)

Requires:

- cfg must specify the logging->enabled option
- If logging->enabled is *True*, cfg must specify:
  - the logging->level option
  - the directories->logs option

translateLevel(*string*)

Parameters

`string`: The string to translate to a logging level

(type=string)

Return Value

A corresponding constant from the `logging` module

(type=integer)

Raises

`Exception` Throw an exception if the given string is not matched

Note: The given string is converted to lowercase and `strip` is applied before any comparisons

3.2 Variables

Name	Description
__package__	**Value:** `'morpher.misc'`

4 Module morpher.misc.section_reporter

Author: Rob Waaser

Contact: robwaaser@gmail.com

Organization: Carnegie Mellon University

Since: November 2, 2011

4.1 Variables

Name	Description
__package__	**Value:** `'morpher.misc'`

4.2 Class SectionReporter

object
└
morpher.misc.section_reporter.SectionReporter

Extends `StatusReporter` with additional functionality for multi-part status bars

`StatusReporter` requires that you know the number of events that will be tracked at the time the status bar is created. However in some cases this information is not completely known. For example, a program might be written to process ten batches of files, but the number of files in the each batch is not known until the previous batch is completed. SectionReporter allows the status bar to be divided into a known number of sections, but the number of events tracked in each section does not need to be known until that section is reached by the status bar. This allows the status bar to display quasi-accurate completion information and remaining time estimates even if the actual information is impossible to determine at that time.

SectionReporter objects can also be reused multiple times by using the `start` method, which essentially resets the counter. The usage pattern is:

```
rep = SectionReporter(2)
rep.start()
rep.startSection(1, 10)
...call rep.pulse() ten times....
rep.endSection()
rep.startSection(2, 3)
...call rep.pulse() three times
rep.endSection()
```

Warning: The status bar assumes that no other output is sent to the console in dynamic update mode and will not display correctly otherwise

See Also: `StatusReporter` is the base class for this class

4.2.1 Methods

__init__(*self*, *numsections*)

Initializes a new object wrapping an underlying `StatusReporter` object using default settings

Parameters

`numsections`: The total number of sections tracked by the status bar

(type=integer)

Overrides: object.__init__

start(*self*, *msg*=`' Status:'`)

Resets the internal counters, prints a message, and prints the empty status bar.

Parameters

`msg`: The message to print just above the status bar, default is "Status:"

(type=string)

Note: The elapsed time is calculated from the last time this method was called for this object

startSection(*self*, *section*, *numevents*)

Sets the current section to the given section number and sets the total number of events tracked by this section

The variable "curevents" is dynamically scaled at this time as if all previous sections had also tracked the same number of events

Parameters

`section`: The index of the section to start, beginning from 1.

(type=integer)

`numevents`: Total number of events tracked by this section.

(type=integer)

pulse(*self*, *events*=`1`)

Increments the number of events completed by the given amount, or 1 by default, then reprints the status bar.

Parameters

`events`: The number of events to increment the counter by, default is 1

(type=integer)

Note: The status bar will not actually reflect this section as being 100 percent complete until `endSection` is called.

endSection(*self*)

Ends the current section, correcting the status bar to reflect exactly *(cursection/numsections)*100* percent completion

Inherited from object

__delattr__(), __format__(), __getattribute__(), __hash__(), __new__(), __reduce__(), __reduce_ex__(), __repr__(), __setattr__(), __sizeof__(), __str__(), __subclasshook__()

4.2.2 Properties

Name	Description
Inherited from object	
__class__	

4.2.3 Instance Variables

Name	Description
curevents	The total number of events that have completed, across all sections - dynamically scaled when a new section is entered as if all previous sections were composed of the same number of total events
cursection	The current section index, starting from 1
curtotal	The total number of events tracked by the current section
numsections	The total number of sections making up the status bar
reporter	The encapsulated `StatusReporter` object used to print the status bar itself.

5 Module morpher.misc.status_reporter

Author: Rob Waaser

Contact: robwaaser@gmail.com

Organization: Carnegie Mellon University

Since: November 1, 2011

5.1 Variables

Name	Description
__package__	**Value:** `'morpher.misc'`

5.2 Class StatusReporter

object
└
morpher.misc.status_reporter.StatusReporter

Used for displaying a status bar and dynamically updating it on the command line

Tracks program progress by keeping an internal counter that can be incremented by the user, and displays an equivalent status bar and estimated completion time on the command line. When an instance is created the user specifies how many "events" must be completed before the program is considered to have finished. The user can then update the number of "events completed" frequently as the program runs, and the status bar will be dynamically updated on the command line along with an estimated completion time, calculated based on the number of events completed so far, the elapsed time, and the number of events left to complete.

StatusReporter objects can also be reused multiple times by using the `start` method, which essentially resets the counter.

Warning: The status bar assumes that no other output is sent to the console in dynamic update mode and will not display correctly otherwise

See Also: `SectionReporter` extends this class with additional capability

5.2.1 Methods

__init__(*self*, *total*=`100`, *size*=`20`, *dynamic*=`True`, *estimate*=`True`)

Initializes a new object with the given settings which can be reused multiple times for printing status bars.

Parameters

`total:` The total number of events that need to be completed, default is 100

(type=integer)

`size:` The number of units in the displayed status bar, default is 20

(type=integer)

`dynamic:` Enables dynamic updating of the same displayed status bar instead of reprinting on a new line, default is *True*

(type=boolean)

`estimate:` Enables displaying the estimated time remaining, default is *True*

(type=boolean)

Overrides: object.__init__

start(*self*, *msg*=`' Status:'`)

Resets the internal counters, prints a message, and prints the empty status bar.

Parameters

`msg:` The message to print just above the status bar, default is "Status:"

(type=string)

Note: The elapsed time is calculated from the last time this method was called for this object

pulse(*self*, *events*=`1`)

Increments the number of events completed by the given amount, or 1 by default, then reprints the status bar.

Parameters

`events:` The amount to increment the completed events by, default is 1

(type=integer)

correct(*self*, *events*)

Sets the number of completed events to the given number

Parameters

`events`: The number to set the completed event counter to

(type=integer)

done(*self*)

Triggers immediate completion for this status bar, just as if all the events had completed normally

printBar(*self*)

Formats and prints the status bar to stdout.

When displayed the status bar should appear similar to:

```
Status:
[====          ]  25% Estimated  1 min 30 sec remaining.....
```

If dynamic updating is set, the last line is erased and rewritten each time the bar is updated; otherwise it is reprinted on the next line.

Inherited from object

__delattr__(), __format__(), __getattribute__(), __hash__(), __new__(), __reduce__(), __reduce_ex__(), __repr__(), __setattr__(), __sizeof__(), __str__(), __subclasshook__()

5.2.2 Properties

Name	Description
Inherited from object	
__class__	

5.2.3 Instance Variables

Name	Description
current	The number of units to display in the status bar
dynamic	Boolean determining if the status bar should be erased and reprinted on the console instead of printed on a new line for each update

continued on next page

Name	Description
estimate	Boolean determining if the estimated time remaining should be displayed along with the status bar
events	The total number of events that have occurred
maxlen	The maximum length that the status bar can print on a line
size	The number of units in the displayed status bar
starttime	The time that the `start` method was called
total	The total number of events the statusbar is tracking

Index