Pyitect Documentation

Release 2.0.0

Benjamin Ryex Powers

August 25, 2015

Contents

A architect inspired plugin framework for Python 3 and Python >= 2.6

A simple Framework that Provides the facility to load Component form plugins.

Also binds a simple event system to bind functions to events on the system.

CHAPTER 1

Links

- GitHub: https://github.com/Ryex/pyitect
- PyPi: https://pypi.python.org/pypi/pyitect
- Travic-CI: https://travis-ci.org/Ryex/pyitect
- Docs: http://pyitect.readthedocs.org/en/latest/

Contents:

1.1 pyitect package

1.1.1 Submodules

pyitect.imports module

This is the shadow module used as a namespace for providing component to loading plugins during import

1.1.2 Module contents

Pyitect is a pluginframe work

class pyitect.**Version**
Version class imported directly from *semantic_version*

see the python-semanticversion project for more information.

class pyitect.**Spec**
Spec class imported directly from *semantic_version*

see the python-semanticversion project for more information.

class pyitect.**System**(*config*, *enable_yaml=False*)
A plugin system

It can scan dir trees to find plugins and their provided/needed components, and with a simple load call chain load all the plugins needed.

The system includes a simple event system and fires some events internal, here are their signatures:

'plugin_found': (path, plugin) path (str): the full path to the folder containing the plugin

plugin (str): plugin version string (ie 'plugin_name:version')

'plugin_loaded': (plugin, plugin_required, component_needed) plugin (str): plugin version string (ie 'plugin_name:version')

plugin_required (str): version string of the plugin that required the loaded plugin (version string ie 'plugin_name:version')

component_needed (str): the name of the component needed by the requesting plugin

'component_loaded': (component, plugin_required, plugin_loaded) component (str): the name of the component loaded

plugin_required (str, None): version string of the plugin that required the loaded component (version string ie 'plugin_name:version') (might be None)

plugin_loaded (str): version string of the plugin that the component was loaded from (version string ie 'plugin_name:version')

Pyitect keeps track of all the instances of the System class in *System.systems* which is a map of object id's to instances of System.

config
dict

A mapping of component names to version requirements

plugins
dict

A mapping of the plugins the system knows about. Maps names to *dicts* of `Version` s mapped to `Plugin` config objects

components
dict

A mapping of `Component.key()` s to loaded component objects

component_map
dict

A mapping of components the system knows about. Maps names to *dicts* of `Version` s mapped to `Component` config objects

loaded_plugins
dict

A mapping of `Plugin.key()` s to loaded plugin module objects

enabled_plugins
list

A list of `Plugin.key()` s of enabled plugins

using
list

A List of `Component.key()` s loaded by the system

events
dict

A mapping of event names to lists of callable objects

add_plugin (*path*)
Adds a plugin form the provided path

Parameters **path** (*str*) – path to a plugin folder

Rasies: PyitectError: If no plugin exists at path PyitectDupError: if you try to add the same plugin twice

bind_event(*event*, *function*)
Bind a callable object to the event name

a simple event system bound to the plugin system, bind a function on an event and when the event is fired all bound functions are called with the **args* and ***kwargs* passed to the fire call

Parameters

- **event** (*str*) – name of event to bind to
- **function** (*callable*) – Boject to be called when event fires

enable_plugins(**plugins*)
Take one or more *Plugin* s and map it's components

Takes a plugins metadata and remembers it's provided components so the system is awear of them

Parameters

- **plugins** (plugins) – One or more plugins to enable.
- **argument can it self be a list or map of** (*Each*) – class:*Plugin*
- **or a plain** (*objects*) – class:*Plugin* object

Raises

- TypeError – If you try to pass a non *Plugin* object
- PyitectDubError – If you try to enable a plugin
- **that provides duplicate conponent** –
- *PyitectOnEnableError* – If There was an error in the on_enable
- *PyitectLoadError* – If there was an error loading a plugin
- **to call it's on_enable** –

fire_event(*event*, **args*, ***kwargs*)
Call all functions bound to the event name

and pass all extra **args* and ***kwargs* to the bound functions

Parameters **event** (*str*) – name of event to fire

get_plugin_module(*plugin*, *version=None*)
Fetch the loaded plugin module

if *version* is None searches for the highest version number plugin with it's module loaded if it can't find anything it raises a runtime error

Parameters

- **plugin** (*str*) – name of plugin to find
- **version** (*None, str, Version*) – if provided load a spesfic version

Returns loaded module object

Raises

- TypeError – if provideing a version that is not either a *str* or
- a – class:*Version*
- *PyitectError* – if the Plugin can't be found

- *PyitectLoadError* – plugin module is not loaded yet

is_plugin(*path*)
Test a path to see if it is a *Plugin*

Parameters **path** (*str*) – path to test

Returns: true if there is a plugin in the folder pointed to by path

iter_component_providers(*comp*, *subs=False*, *vers=False*, *reqs='*'*)
An iterater function to interate providers of a component

Takes a conponent name and yeilds providers of the conponent

if *subs* is *True* yeilds providers of subtypes too

if *vers* is *True* yeilds all version of the provider not just the highest

reqs is a version requirement for the providers to meet. Defaults to any version

Parameters

- **comp** (*str*) – component name to use as a base
- **subs** (*bool*) – should subtypes be yeilded too?
- **vers** (*bool*) – should all version be yeilded not just the highest?
- **reqs** (*str, list, tuple*) – version spec string or list there of
- **items are passed to a Spec** (*all*) –

Raises TypeError – if *comp* or *reqs* are passed wrong

iter_component_subtypes(*component*)
An iterater function to interate all known subtypes of a component

Takes a conponent name and yeilds all known conponent names that are subtypes not including the conponent name

Parameters **conponent** (*str*) – the conponent name to act as a base

Raises

- TypeError – if ‘‘component‘ is niether a
- :class – *Component* instance nor a string

load(*component*, *requires=None*, *request=None*, *bypass=False*)
Load and return a component object

processes loading and returns the component by name, chain loading any required plugins to obtain dependencies. Uses the config that was provided on system creation to load correct versions, if there is a conflict throws a run time error. bypass lets the call bypass the system configuration

Parameters

- **component** (*str*) – name of component to load
- **requires** (*dict, None*) – a mapping of component names
- **version requierments to use during the load** (*to*) –
- **request** (*str, None*) – the name of the requesting plugin.
- **if not requested** (*None*) –
- **bypass** (*bool*) – ignore the system configured version requierments

Returns the loaded component object

Raises

- `TypeError` – if thigns get passed worng
- *`PyitectLoadError`* – if there is an exception druing load

load_plugin(*plugin*, *version*, *requires=None*, *request=None*, *comp=None*)
Takes a plugin name and version and loads it's module

finds the stored Plugin object takes a Plugin object and loads the module recursively loading declared dependencies

Parameters

- **plugin** (*str*) – plugin name
- **version** (*str, Version*) – version to load
- **requires** (*dict, None*) – a mapping of component names
- **version requierments to use during the load** (*to*) –
- **request** (*str, None*) – name of the version string of the plugin
- **requested a component from this plugin.** (*that*) –
- **if not requested.** (*None*) –
- **comp** (*str*) – name of the component needed by teh requesting plugin.
- **if not requested.** –

Returns the loaded module object

Raises

- `TypeError` – if things get passed worng
- *`PyitectLoadError`* – if there is an exception during the load

resolve_highest_match(*component*, *plugin*, *spec*)
resolves the latest version of a component with requirements,

takes in a component name and some requierments and gets a valid plugin name and its highest version

Parameters

- **component** (*str*) – a component name
- **plugin** (*str*) – a plugin name if it's empty we default to alphabetical order
- **spec** (Spec) – a SemVer version spec

Raises `TypeError` – if somthing isn't the right type

search(*path*)
Search a path (dir or file) for a plugin in the case of a file it searches the containing dir.

Parameters **path** (*str*) – the path to search

systems = []
A list of all *`System`* instances

unbind_event(*event*, *function*)
Remove a function from an event

removes the function object from the list of callables to call when event fires. does nothing if function is not bound

Parameters

- **event** (*str*) – name of event bound to
- **function** (*callable*) – object to unbind

class pyitect.**Plugin**(*config*, *path*)

An object that can hold the metadata for a plugin

like its name, author, verison, and the file to be loaded ect. also stores the path to the plugin folder and provideds functionality to load the plugin module and run its *on_enable* function

name

str

plugin name

author

str

plugin author

version

Version

plugin vesion

file

str

relative path to the file to import to load the plugin

consumes

dict

a listing of the components consumed

provides

dict

a listing of the components provided

on_enable

None, str

either *None* or a str doted name of a function in the module

path

str

an absolute path to the plugin folder

module

None, object

either *None* or the modlue object if the plugin has been loaded already

get_version_string()

returns a version string

has_on_enable()

returns *True* if it has an *on_enable* attribute that's not None

key()
return a key that can be used to identify the plugin

Returns (name, author, version, path)

Return type tuple

load()
loads the plugin file and returns the resulting module

Raises

- *PyitectLoadError* – If there was a problem loading a plugin module
- *PyitectNotProvidedError* – If component is not provided

run_on_enable()
runs the function in the 'on_enable' if set

Raises

- TypeError – if the on_enable property is set wrong
- *PyitectOnEnableError* – if there is an exception
- **acessing or calling the on_enable function** –
- *PyitectLoadError* – If the module object is not loaded yet

class pyitect.**Component**(*name*, *plugin*, *author*, *version*, *path*)
An object to hold metadata for a spesfic instance of a component

Holds the metadata needed to identify a instance of a component provided by a plugin

name
str

the component name provided

plugin
str

the name of the providing plugin

author
str

the author of the providing plugin

version
Version

the verison of the providing plugin

path
str

a doted name path to the component object from the top of the plugin module

key()
returns a key to identify this component

Returns (name, plugin, author, version, path)

Return type tuple

pyitect.**get_system**()
Fetch the global system instance

Raises *PyitectError* – If the system isn't built yet

pyitect.**build_system**(*config*, *enable_yaml=False*)
Build a global system instance

Parameters

- **config** (*dict*) – A mapping of component names to version requirements
- **enable_yaml** (*bool*) – Should the system support yaml config files?

Raises *PyitectError* – if the system is already built

pyitect.**destroy_system**()
destroy the global system instance

does nothing if the system isn't built

pyitect.**issubcomponent**(*comp1*, *comp2*)
Check if comp1 is a subtype of comp2

Returns whether the Component passed as comp1 validates as a subtype of the Component passed as comp2.

if strings are passed as either peramater they are treated as Component names. if a Component instance is passed it's *name* property is pulled.

Parameters

- **comp1** (*str, Component*) – The Component or component name to check
- **comp2** (*str, Component*) – The Component or component name to compair to

pyitect.**get_unique_name**(**parts*)
Generate a fixed lenght unique name from parts

takes the parts turns them into strings and uses them in a sha1 hash

used internaly to ensure module object for plugins have unique names like so

get_unique_name(plugin.author, plugin.get_version_string())

Returns name hash

Return type str

pyitect.**gen_version**(*version_str*)
Generates an *Version* object

takes a SemVer string and returns a *Version* if not a proper SemVer string it coerces it

Parameters **version_str** (*str*) – version string to use

pyitect.**expand_version_req**(*requires*)
Take a requierment and return the Spec and the plugin name

takes a requierment and pumps out a plugin name and a SemVer Spec requires is either a string of the form *("", "*", "plugin_name", plugin_name:version_spec)*

or a mapping with *plugin* and *spec* keys like so *{"plugin": "plugin_name", "spec": ">=1.0.0"}* the spec key's value can be a string of comma seperated version requierments or a list of strings of the same

Parameters **requires** (*str, mapping*) – string or mapping object with *plugin* and *spec* keys

Examples

```
>>> expand_version_req("")
('', <Spec: (<SpecItem: * ''>,)>)
>>> expand_version_req("*")
('', <Spec: (<SpecItem: * ''>,)>)
>>> expand_version_req("plugin_name")
('plugin_name', <Spec: (<SpecItem: * ''>,)>)
>>> expand_version_req("plugin_name:>=1.0.0")
('plugin_name', <Spec: (<SpecItem: >= Version('1.0.0')>,)>)
>>> expand_version_req("plugin_name:>=1.0.0,<2.0.0")
('plugin_name', <Spec: (SpecItems... >= 1.0.0, < 2.0.0 )>)
>>> expand_version_req({"plugin": "plugin_name", "spec": ">=1.0.0"})
('plugin_name', <Spec: (<SpecItem: >= Version('1.0.0')>,)>)
```

Raises

- `ValueError` – when the requierment is of a bad form
- `TypeError` – when the requiers objt is not a string or mapping

exception `pyitect.`**`PyitectError`**(**args*, ***kwargs*)
Wraps Exceptions for Chained trace backs

As Pyitect is intended for use across pyhton 2 and 3 a way was needed to Ensure that exceptions caused during the import of plugin modules tell *why* that import failed insed of just *failed to import 'bla'*

This code is a modifed version of a *CausedException* class posed to ActiveState back in Sep. 2012 Licensed under MIT license

the ability handle trees of exceptions was removed

http://code.activestate.com/recipes/578252-python-exception-chains-or-trees/?in=user-4182236

exception `pyitect.`**`PyitectNotProvidedError`**(**args*, ***kwargs*)
Raised if a conponent is not provided

exception `pyitect.`**`PyitectNotMetError`**(**args*, ***kwargs*)
Raised if requierments are not met

exception `pyitect.`**`PyitectLoadError`**(**args*, ***kwargs*)
Raises if a plugins module is not yet loaded or fais to load

exception `pyitect.`**`PyitectOnEnableError`**(**args*, ***kwargs*)
Raised if and on_enable call failes

exception `pyitect.`**`PyitectDupError`**(**args*, ***kwargs*)
Raised if you try to add a duplicate plugin or duplicate component provider

1.2 Creating Plugins

Contents:

1.2.1 What is a Plugin?

A plugin to pyitect is simply a folder with a *.json* config file of the same name as the folder inside. If you have yaml support enabled the extensions *.yaml* and *.yml* are also available

```
/Im-A-Plugin
    Im-A-Plugin.json
    file.py
```

```
/Im-A-Plugin2
    Im-A-Plugin2.yaml
    file.py
```

```
/Im-A-Plugin3
    Im-A-Plugin3.yml
    file.py
```

A plugin has a name, a version, an author, a module or package, and it provides Components used to build your application. a component is simply an object which can be accessed from the imported module a plugin's config file provides information about the plugin as well as lists components it provides and components it needs on load

Here's an example, most fields are mandatory but the consumes and provides CAN be left as empty containers

```
{
    "name": "Im-A-Plugin",
    "author": "author_name",
    "version": "0.0.1",
    "file": "file.py",
    "on_enable": "on_enable_func",
    "consumes": {
        "foo" : "*"
    },
    "provides": {
        "Bar": ""
    }
}
```

Here is the same file in yaml

```
name: Im-A-Plugin
author: author_name
version: 0.0.1
file: file.py
on_enable: on_enable_func # optional, runs this function when the plugin is enabled
consumes:
    foo: '*'
provides:
    Bar: ''
```

Version numbers should conform to conformed to Semantic Versioning meaning that they should have a major, minor, and patch number like so: *major.minor.patch-prereleace+buildifo*

- **name** -> the name of the plugin (No spaces)
- **author** -> the author of the plugin
- **version** -> a version for the plugin, a string that conformes to SemVer
- **file** -> a path to a function that will be called form the imported module after the plugin is loaded
- **consumes** -> a mapping of needed component names to version requierments, empty string = no requirement
- **provides** -> a mapping of provided component names to paths from the imported module, empty string = path is component name

1.2.2 Version Requirements

A plugin can provide version requirements for the components it's importing. they take two forms, a version string or a version mapping.

A version string is formatted like so

```
plugin_name:<version_requirements>
```

Both parts are optional and an empty string or a string containing only a '*' means no requirement. If there is no requirement specified then the highest available version will be selected from the first provider in alphabetical order.

if the version requirement is not give or given as * but the plugin name is then the highest available version will be selected from the names plugin

A version requirement is a logical operator paired with a version number. Any number of requirements can be grouped with commas.

Version numbers in requirements should also follow Semantic Versioning

Version requirement support is provided by the python-semanticversion project. specifically the *Spec* class. More documentation can be found here.

Here are some examples of a version string

```
""  // no requirement
"*" // no requirement
"FooPlugin" // from this plugin and no other, but any version
"FooPlugin:*" // from this plugin and no other, but any version
"FooPlugin:==1" // from this plugin and no other, version 1.x.x
"FooPlugin:==1.0" // 1.0.x
"FooPlugin:==1.0.1" // version 1.0.1 or any post release
"FooPlugin:==1.0.1-pre123" // 1.0.1-pre123 -> this exact version
"FooPlugin:==1.2" // 1.2.x and any pre/post/dev release
"FooPlugin:>1.0" // greater than 1.0
"FooPlugin:>=1.2.3" // greater than or equal to 1.2.3
"FooPlugin:<=2.1.4" // less than or equal to 2.1.4
"FooPlugin:>1.0,<2.3" // greater than 1.0 and less than 2.3
"FooPlugin:>1.0,<=2.0,!=1.3.17" // between V1.0.x and V2.0.x but not V1.3.17
```

Version requirements can also be given a a mapping. The mapping must contain the keys *plugin* and *spec* but this can allow for your requirement specification to be more clear.

Here is an example:

```
{
    "name": "Im-A-Plugin2",
    "author": "author_name",
    "version": "0.0.1",
    "file": "file.py",
    "consumes": {
        "foo" : {
            "plugin": "special_plugin_name",
            "spec": ">1.0,<=2.0,!=1.3.17"
        }
    },
    "provides": {
        "Bar": ""
    }
}
```

The *spec* key can also be a list of version specifications

```
{
    "consumes": {
        "foo" : {
            "plugin": "special_plugin_name",
            "spec": [">1.0", "<=2.0,!=1.3.17"]
        }
    }
}
```

1.2.3 Letting Plugins Access Consumed Components

inside your plugin files you need to get access to your consumed components right? Here's how you do it.

The plugin can pull it's declared components from *pyitect.imports* during the import of the module or package. *pyitect.imports* gets cleared after the import is done. So, the component imports from *pyitect.imports* should be in the top level of the module, not on demand imports in the code.

if a plugin author needs access to components not declared in the config file for run time use - ie. to load component on the fly - then they will need the system author to provide access to the plugin system instance.

1.2.4 Writing a Plugin

Writing a plugin for pyitect is simple.

First Create a folder to hold your plugin

Second Create a configuration file with the same name as the folder but with an extension. *.json* for a JSON config or *.yaml*/*.yml* for a YAML config

Third Create your python module or package. Your plugin folder can even be your package folder

Forth Write up your config for the plugin

> Point the file attribute to your module file or package. If it's a package point it to the *__init__.py*. It doesn't matter if your module is pure python, byte-code compiled (*.pyc*) or a native extension (*.pyd*, *.so*)

A working plugin looks something like the following:

Folder Structure

```
/Im-A-Plugin
    Im-A-Plugin.json
    file.py
```

Im-A-Plugin.json

```
{
    "name": "plugin_name",
    "author": "author_name",
    "version": "0.1.0",
    "file": "<relative_path>",
    "on_enable": "<optional_function_path>",
    "consumes": {
        "foo" : "*"
    },
    "provides": {
```

```
        "Bar": ""
    }
```

file.py

```
#file.py
from pyitect.imports import foo

class Bar(object):
    def __init__():
        foo("it's a good day to be a plugin")
```

1.3 Setting up a Plugin system

Setting up a plugin system is dead simple.

First create an instance of the *System* class

```
from pyitect import System
system = System()
```

The system class constructor takes two arguments, a configuration mapping and a *yaml* flag

The config mapping allows you to provide default requirements for components so if a call is made to *system.load()* with no requirements of it's own the requirements from the passed config are used.

The *yaml* flag of course enables *yaml* support for the plugin system. allowing configuration file to be written in yaml. *yaml* support is not enabled by default because it requires the PyYAML library.

Next add some plugins to the system.

This can be done either by using the *system.search()* function to recursively search a directory for plugins.

Or added manualy by providing the path to a plugin folder to the *system.add_plugin()* function of your system instance.

```
system.search("path/to/your/plugins/tree")
system.add_plugin("paht/to/a/plugin/folder")
```

Now that you have some plugin you still have to enable them.

Enabling a plugin maps out the components it provides and make them available for loading by the plugin system. Tt does not load the plugin module or package unless there is an 'on_enable' property in its configuration.

In which case, after the component are mapped and the plugin is enabled, the plugin module or package is loaded and an attempt is made to follow the path given to the *on_enable* configuration property to a callable object (ie. function) from the top level of the module or package and it is called passing only the *Plugin* configuration object for the plugin.

To enable a plugin you needs it's *Plugin* instance. these can be accessed from *system.plugins*

a simple way to get a list of them would be.

```
plugins = [system.plugins[n][v] for n in system.plugins for v in system.plugins[n]]
```

After you have your list of Plugin objects you can filer it how you want to enable only the plugins you want to. When your ready.

```
system.enable_plugins(plugins)
```

***enable_plugins* can take multiple objects and any individual** can by a iterable or map of *Plugin* objects.

After you have some plugins enabled loading a provided component is as easy as

```
Bar = system.load("Bar")
```

The general idea is to create a system, search some path or paths for plugins and then enable them.

A plugin system can not be created without first creating an instance of the System class.

1.3.1 Global System

If you dont want to manage your plugin system instance yourself it is possible to have the pyitect module manage your plugin system for you. Simply use the *pyitect.build_system()* function to construct your plugin system inside pyitect. To later fetch your plugin system instance use *pyitect.get_system()*. To clean up and remove the existing system use *pyitect.destroy_system()*.

1.3.2 'on_enable' Property

plugins can specify an *on_enable* property in their configuration. This is a doted name path to a function that is is executed right after a plugin is enabled and its components have been mapped. This allows for special cases where enabling a plugin requires more than just making it's components available to be imported. For example is there is some system setup to be done.

```
pyitect.build_system(config, enable_yaml=False)
system = pyitect.get_system()
# ... do stuff
# end program / need fresh system?
pyitect.destroy_system()
```

1.3.3 Loading Plugins

Plugins are loaded on demand when a component is loaded via

```
system.load("<component name>")
```

a plugin can also be explicitly loaded via

```
system.load_plugin(plugin, version)
```

where *plugin* is the plugin name and *version* is the version

1.3.4 Tracking loaded Components

Pyitect tracks used components at anytime *system.using* can be inspected to find all components that have been requested and from what plugins they have been loaded along with versions.

system.using is a list of *component.key()* s

```
>>> system.using
{
    'component1' : {
        'plugin1`: ['1.0.2']
    },
    'special_component1' : {
```

```
        'special_plugin1': ['0.1.3'],
        'special_plugin2': ['0.2.4', '1.0.1-pre3']
    }
}
```

Pyitect also tracks enabled plugins `system.enabeled_plugins` is a mapping of plugin names to a mapping of versions to *`Plugin`* objects.

Like so

```
>>> system.enabeled_plugins
{
    "special_plugin1" : {
        "Version('1.0.0')": Plugin('special_plugin1:1.0.0')
    }
}
```

1.4 Useing Events

The plugin system also includes a simple event system bound to the *`system`* object, it simply allows one to register a function to an event name and when *`system.fire_event`* is called it calls all registered functions passing the extra **args* and ***kwargs* to them.

pyitect fires some events internally so that you can keep track of when the system finds and loads plugins.

1.4.1 Using Events

Pyitect supplies three methods for dealing with events

`System.bind_event`

```
system.bind_event('name', Function)
```

Binds *Function* to the event *'name'*. when an event of *'name'* is fired the function will be called wall all extra parameters passed to the *fire_event* call.

`System.unbind_event`

```
system.unbind_event('name', Function)
```

Removes *Function* form the list of functions to be called when the event is fired

`System.fire_event`

```
system.fire_event('name', *args, **kwargs)
```

Fires the event *'name'*, calling all bound functions with **args* and ***kwargs*

1.4.2 Events Fired Internally

plugin_found

A function bound to this event gets called every time a plugin is found during a search called an example is provided.

Example function to bind:

```
def onPluginFound (path, plugin):
    """
    path (str): the full path to the folder containing the plugin
    plugin (str): plugin version string (ie 'plugin_name:version')
    """
    print("plugin `%s` found at `%s`" % (plugin, path))
```

component_mapped

When a plugin is enabled it's components are mapped out, this event is fired ever time that happens.

Example function to bind:

```
def onComponentMapped (component, plugin, version):
    """
    component (str): the component name
    plugin (str): plugin name
    version (Version): the plugin version string less the plugin name
    """
    print("component `%s` mapped form `%s@%s`" % (component, plugin, version))
```

plugin_loaded

A function bound to this event is called every time a new plugin is loaded during a component load.

Example function to bind:

```
def onPluginLoad (plugin, plugin_required, component_needed):
    """
    plugin (str): plugin version string (ie 'plugin_name:version')
    plugin_required (str): version string of the plugin that required the loaded plugin (version stri
    component_needed (str): the name of the component needed by the requesting plugin
    """
    print("plugin `%s` was loaded by plugin `%s` during a request for the `%s` component" % (plugin,
```

component_loaded

A function bound to this event is called every time a component is successfully loaded example:

Example function to bind:

```
def onComponentLoad (component, plugin_required, plugin_loaded):
    """
    component (str): the name of the component loaded
    plugin_required (str): version string of the plugin that required the loaded component (version s
    plugin_loaded (str): version string of the plugin that the component was loaded from (version sti
    """
    print("Component `%s` loaded, required by `%s`, loaded from `%s`" % (component, plugin_required,
```

1.5 ChangeLog

1.5.1 v2.0.0 (2015-8-25)

- Large API incompatible update
- supports Python 2.6+
- now uses *pyitect.imports* for import time plugin loading
- version postfixes are replaced with component subtypes
- Uses SemVer processing via python-semanticversion project
- module overlap is prevented with unique model names in *sys.modules*
- no import modes, uses *imp* module for everything >= 3.3 and *importlib* for 3.4+
- ability to store global system instance in *pyitect* module
- support YAML for plugin configuration files
- Custom Exception classes with exception chain support
- fully fledge docs

1.5.2 v1.1.0 (2015-7-17)

- readme cleanup
- gen_version generates a version 2 tuple
- change on_enable to a callable path in the imported plugin module

1.5.3 v1.0.1 (2015-6-10)

- change out Version mechanism for a local parse method based off of LooseVersion
- update tests to proper unit tests

1.5.4 v1.0.0 (2015-6-9)

- change from parse_version from setuptools to LooseVersion in distutils

1.5.5 v0.9.2 (2014-9-28)

- Ensure plugin configuration json file is closed @svisser

1.5.6 v0.9.1 (2014-9-28)

- files loaded with *exec* give proper file path
- proper trace back given when component fail to load (even when it's a recursion error)
- add *component_mapped* event

1.5.7 v0.9.0 (2014-9-27)

- add *get_plugin_module* method

1.5.8 v0.8.0 (2014-9-27)

- Added ability to run code when a plugin is enabled via *"on_enable"* property

1.5.9 v0.7.2 (2014-9-23)

- Fix name error in unbind and fire event commands

1.5.10 v0.7.0 (2014-9-21)

- plugins found with *System.search* are no longer auto enabeled
- use *System.enable_plugins(<mapping>|<iterable>|<Plugin>)* to enable plugins from *System.plugins*
- added *Plugin* class to main namespace

1.5.11 v0.6.2 (2014-9-13)

- relative imports now work so long as the target file for loading is named *__init__.py* to trigger python to treat the plugin folder as a package

1.5.12 v0.6.1 (2014-9-13)

- re-factored System.load out to make use of two smaller functions, easyer to maintain
- added plugin loading modes, import for py3.4+ and exec for support of previous python version

1.5.13 v0.5.1 (2014-8-30)

- added ability to provide more than one version of a component in the same plugin with potfix mapping
- event system added, system fire events
- made requirement overwrite system defaults, removed bypass peram
- ittrPluginsByComponent lists potfix mappings too.
- tests updates to test all features
- README update
- This changelog added

1.5.14 v0.1.15 (2014-8-26)

- added ittrPluginsByComponent
- added bypass peram to System.load to bypass system default

1.5.15 v0.1.10 (2014-8-25)

- First public release

CHAPTER 2

Indices and tables

- genindex
- modindex
- search

p

R

S

U

V