Ways Documentation

Release 1.0.0b1

Colin Kennedy

Jun 24, 2018

Contents

Welcome to Ways, an AMS toolkit for Python. For the tired programmer that has to deal with projects with expanding scope and revisions, this tool is for you.

For more information, read on

CHAPTER 1

Main Pages

1.1 How To Install

1.1.1 Install via PyPI

Installing the deployed versions of Ways is recommended. To install Ways, just run:

```
pip install ways
```

1.2 Why use Ways

The problem with writing code isn't actually writing it for the first time. It's changing code later that causes the most issues. Project requirements change or I/O can out of hand. Depending on the changes needed, multiple tools might need to be updated at once.

Dealing with these complexity-scenarios is where Ways truly shines.

1.2.1 Basics

Ways is a Python toolkit which is supported by config files called "Plugin Sheets". This is an example of a relatively simple Plugin Sheet.

```
plugins:
    some_plugin:
        hierarchy: some/hierarchy
        mapping: /path/to/a/{JOB}/here
```

This Plugin Sheet is written using YAML but it can written in Python or JSON, too. The important bit in this Plugin Shet is the "hierarchy" key. The string used there is what we'll use to get Context and Asset objects in other examples.

Note: This page will reference "Context" and "Asset" objects a lot. (`ways.api.Context` and `ways.api.Asset`).

They're both explained in other pages so, for now, just know that Context objects help get plugins and Asset objects add functionality to Context objects.

To make a Context and Asset from the Plugin Sheet that was written earlier, you would use `ways.api.get_context()` and `ways.api.get_asset()`.

```
path = '/path/to/a/job_name/here'
asset = ways.api.get_asset(path, context='some/hierarchy')
asset.get_value('JOB')
# Result: 'job_name'
```

1.2.2 Extend Ways Using Actions

Context and Asset objects have very few methods by default and almost every method just queries information defined in a Plugin Sheet. To actually write methods that use a Context or Asset, we need to define an Action for it.

An Action is any callable object, class or function, that takes at least one argument. The first argument given to an Action will always be the Asset or Context that called it.

There are two ways to create Action objects. Create a class/function and "register" it to Ways or subclass `ways.api.Action`, and Ways will register it for you.

```
# Method #1: Subclass ways.api.Action
class SomeAction(ways.api.Action):

    name = 'some_action'

    def __call__(self, context):
        return 8

    @classmethod
    def get_hierarchy(cls):
        return 'some/hierarchy'

# Method #2: Register a function or class, explicitly
def some_function(obj):
    return 8

def main():
    ways.api.add_action(some_function, name='function', context='some/hierarchy')

# Actually using the Actions
context = ways.api.get_context('some/hierarchy')

context.actions.some_action()
context.actions.function()
```

Actions let the user link Contexts together, manipulate data, or communicate between different APIs.

1.2.3 Mixing Ways with other APIs

Many examples in this page and others use Ways to describe filepaths. This isn't a requirement for Ways, it's just to keep examples simple. The truth is, in practice, if you're using Ways only to deal with filepaths, Ways won't be much better than a database.

But Ways doesn't need to represent paths on disk, Ways can represent anything as long as it can be broken down into a string.

A common situation that comes up in the VFX industry is that tools need to communicate with a filesystem, a database, and some third-party Python API at once.

For example, say an artist published a new version of a texture on a job's database and we wanted to republish a 3D model with those new textures.

(This example assumes a basic understanding of the tools of VFX artists. Example: Maya is a 3D modeling and animation tool and PyMEL is a Python API used in Maya)

```python
import pymel.core as pm
import ways.api


def get_asset(node):
    '''A function to wrap any supported Maya node into a Ways Asset.'''
    class_name = node.__class__.__name__
    context = 'dcc/maya/{}'.format(class_name)
    return ways.api.get_asset({'uuid': node.get_uuid()}, context=context)


node = pm.selected()[0]  # Use the Maya API to get our selected texture
texture = get_asset(node)

# Now use the database to lookup the published versions of the texture
asset = texture.actions.get_database_asset()

# Get the path of the published texture and add it to the local disk
version = asset.actions.get_latest_version()
path = version.actions.get_filepath()

if not os.path.isfile(path):
    print('Syncing: "{path}" from the database.'.format(path=path))
    version.actions.sync()

asset.actions.set_path(path)

# Now we need to find the rig(s) that contain this texture to republish
rig_sets = []
for node_ in pm.sets(query=True):
    try:
        if node_.attr('setType').get() == 'rig':
            rig_sets.append(node_)
    except pm.MayaAttributeError:
        pass

rigs = []
for rig_node in rig_sets:
    rig = get_asset(rig_node)

    if not rig:
        continue

    if rig.actions.contains(texture):
        rig.actions.publish(convert_to='geometry_cache')  # Publish the new version
```

These sort of API mixtures are possible because of the "hierarchy" key mentioned earlier. Each Context knows about their own hierarchy, the hierarchy of its parent Context, and all child Contexts by looking through its hierarchy which you have full control over.

```
plugins:
    database_root:
        # get_database_asset, under the hood, fills in the info in mapping
        # and then returns another Ways Asset with its own set of Actions.
        #
        hierarchy: db/asset
        mapping: db.{SHOT}.{ASSET_NAME}

    # filepath-related plugin
    textures_output:
        hierarchy: job/shot/textures/release
        # This is an example filepath to publish our texture to
        mapping: "{JOB}/{SCENE}/{SHOT}/releases/{ASSET}_v{VERSION}/{texture}"

    # Maya plugins
    node_object:
        hierarchy: dcc/maya
        mapping: "{uuid}"
        mapping_details:
            uuid:
                parse:
                    regex: "[A-Z0-9]{8}-[A-Z0-9]{4}-[A-Z0-9]{4}-[A-Z0-9]{4}-[A-Z0-9]
↪{12}"

    # Texture-related nodes
    file_node:
        hierarchy: "dcc/maya/nodes/File"
```

The above example only works with Maya "File" nodes. If we wanted to support other Maya texture-related nodes, all we'd have to do is add them to this Plugin Sheet and then implement a "set_path" Action for them.

1.2.4 String Querying

A basic use of Ways would be to get data from a file path. Normally you might do something like this to split a path and get its pieces.

```
def get_parts(path):
    return path.split(os.sep)


def get_environment_info(path):
    '''Parse a path of format "/jobs/{JOB}/{SCENE}/{SHOT}/{DISCIPLINE}".'''
    parts = get_parts(path)

    return {
        'JOB': parts[2],
        'SCENE': parts[3],
        'SHOT': parts[4],
        'DISCIPLINE': parts[4],
    }
```

```
path = '/jobs/someJobName_123/shot_name-Info/sh01/animation'
info = get_environment_info(path)
print(info['JOB'])
# Result: 'someJobName_123'
```

Here is the same example, using Ways. Start by making a Plugin Sheet. We'll call this Plugin Sheet "plugin_sheet.yml".

```
plugins:
    foo_plugin:
        hierarchy: job/shot/discipline
        mapping: /jobs/{JOB}/{SCENE}/{SHOT}/{DISCIPLINE}
        path: true
```

Add the path to "plugin_sheet.yml", to your WAYS_DESCRIPTORS environment variable.

```
export WAYS_DESCRIPTORS=/path/to/plugin_sheet.yml
```

This is what using our plugin in Python would look like

```
import ways.api

path = '/jobs/someJobName_123/shot_name-Info/sh01/animation'
asset = ways.api.get_asset(path)
print(asset.get_value('JOB'))
# Result: 'someJobName_123'
```

Now for some bad news - We need our setup to work with Windows. And worse, the Windows-equivalent path for "/jobs/{JOB}/{SCENE}/{SHOT}/{DISCIPLINE}" has a different number of folders so our old function cannot work for both ("Z:\NETWORK\server1\jobs\{JOB}\{SCENE}\{SHOT}\{DISCIPLINE}").

But in Ways, these sort of changes only require a slight change in our Plugin Sheets.

```
plugins:
    windows_root:
        hierarchy: job
        mapping: Z:\\NETWORK\\jobs
        path: true
        platforms:
            - windows
    linux_root:
        hierarchy: job
        mapping: /jobs
        path: true
        platforms:
            - linux
    discipline:
        hierarchy: '{root}/shot/discipline'
        mapping: '{root}/{JOB}/{SCENE}/{SHOT}/{DISCIPLINE}'
        uses:
            - job
```

```python
import ways.api

path1 = '/jobs/someJobName_123/shot_name-Info/sh01/animation'
asset1 = ways.api.get_asset(path1)
print(asset1.get_value('JOB'))
# Result on Linux: 'someJobName_123'

path2 = r'Z:\NETWORK\jobs\someJobName_123\shot_name-Info\sh01\animation'
asset2 = ways.api.get_asset(path2)
print(asset2.get_value('JOB'))
# Result on Windows: 'someJobName_123'
```

This works because the "discipline" plugin key uses "job" and "job" is defined differently for each OS. To support other operating systems, you write once for each OS and just append any information you need onto it.

String Parsing

Now our project needs to be able to query the "Info" part from SCENE because "Info" is useful to us.

If we're doing a non-Ways solution, like using built-in Python functions or regex, your solution may look something like this:

```python
def get_scene_info(scene):
    '''str: Get the "Info" part of some scene.'''
    return scene.split('-')[-1]

path = '/jobs/someJobName_123/shot_name-Info/sh01/animation'
info = get_environment_info(path)
print(get_scene_info(info['SCENE']))
# Result: 'Info'
```

Using "split('-')" is definitely not ideal because we're forcing a specific convention on the code that will need to be consistent for all of our other tools.

We could make "-" a global variable or read in from a config file and that will help but, either way, getting "Info" becomes a a very granular task.

Imagining what kinds of paths that our program expects without documentation becomes more difficult, as well.

Lets tackle the same problem, using Ways.

```yaml
plugins:
    windows_root:
        hierarchy: job
        mapping: Z:\\NETWORK\\jobs
        path: true
        platforms:
            - windows
    linux_root:
        hierarchy: job
        mapping: /jobs
        path: true
        platforms:
            - linux
    discipline:
        hierarchy: '{root}/shot/discipline'
        mapping: '{root}/{JOB}/{SCENE}/{SHOT}/{DISCIPLINE}'
        mapping_details:
            SCENE:
                mapping: "{SCENE_PREFIX}-{SCENE_INFO}"
        uses:
            - job
```

```
import ways.api

path = '/jobs/someJobName_123/shot_name-Info/sh01/animation'
asset = ways.api.get_asset(path)
print(asset.get_value('SCENE_INFO'))
# Result: 'Info'
```

Between the previous example and this one, only 3 new lines were added.

```
mapping_details:
    SCENE:
        mapping: "{SCENE_PREFIX}-{SCENE_INFO}"
```

There's a lot more to learn about parsing. Ways can handle querying missing data or integrate other parse engines like regex and glob. Most of those topics are pretty dense so lets skip it for now. But, if you want to know more, you can skip ahead to *Parsing Plugins*.

1.2.5 Adding Existing AMS

Most likely, Ways is not the first AMS (Asset Management System) tool you've used. Chances are, you have your own AMS that you'd like to keep using. Ways can partially integrate existing objects into its own code to help tie into existing systems.

```
class MyAssetClass(object):

    '''Some class that is part of an existing AMS.'''

    def __init__(self, info, context):
        super(MyAssetClass, self).__init__()
        # ... more code ...

def main():
    ways.api.register_asset_class(MyAssetClass, context='some/hierarchy')

asset = ways.api.get_asset({}, context='some/hierarchy')
# Result: <MyAssetClass>
```

Now when you run "get_asset", the function will return MyAssetClass. For more information on register_asset_class, check out *Asset Swapping*.

1.2.6 Dealing With Revised Projects

Imagine that you're working on a tool that publishes images to a database. Because you were only working for yourself, you made a function to parse your path:

(Example path: "/jobs/{JOB}/{SCENE}/{SHOT}/elements/{NAME}/{VERSION}/{LAYER}/{SEQUENCE_NAME}" "/jobs/fooJob/fooScene/sh01/elements/frame_Render/v001/beauty/file_sequence.####.tif")

```
def get_sequence_info(path):
    '''Parse a path like get_environment_info.'''
    # ... get the info ...

def publish(info):
    '''Publish to the database with our info.'''
    # Do the publish to our database ...

path = "/jobs/{JOB}/{SCENE}/{SHOT}/elements/frame_Render/v001/beauty/file_sequence.###
↪#.tif"
info = get_sequence_info(path)
info['path'] = path

publish(info)
```

Lets just pretend for a moment that this example did everything we needed to do. Maybe get_sequence_info is some regex or another parser. The point is that, whatever the solution it, it's good enough for the tool that you're writing.

If we used Ways, this is what the same example could look like.

```
plugins:
    linux_root:
        hierarchy: job
        mapping: /jobs
        path: true
    element:
        hierarchy: '{root}/shot/element'
        mapping: '{root}/{JOB}/{SCENE}/{SHOT}/elements'
        uses:
            - job
    sequence_bit:
        hierarchy: '{root}/rendered/sequence'
        mapping: '{root}/{NAME}/{VERSION}/{LAYER}/{SEQUENCE_NAME}'
        uses:
            - job/shot/element
```

Now that we've made the plugins needed for our path, we make an Action object to do the publish.

```
class PublishAction(ways.api.Action):

    name = 'publish'

    @classmethod
    def get_hierarchy(cls):
        return 'job/shot/element'

    def __call__(self, info):
        '''Publish to the database with our info.'''
        # ... do the publish ...
```

```
path = '/jobs/fooJob/fooScene/sh01/elements/frame_Render/v001/beauty/file_sequence.###
↪#.tif'
asset = ways.api.get_asset(path)
asset.actions.publish()
```

Another developer on your team may have developed a tool that depends on those published images too but their tool uses very different paths from yours and you are being asked to accomodate those paths to.

You've been putting files in

"/jobs/{JOB}/{SCENE}/{SHOT}/elements/{NAME}/{VERSION}/{LAYER}/{SEQUENCE_NAME}"

but the other developer has been putting similar files in

"/jobs/{JOB}/{SCENE}/{SHOT}/elements/plates/houdini/{NAME}_{VERSION}/{VERSION}/{LAYER}/file_sequence.####.tif"

Now you're in a bad situation. The other developer is adding files in a completely different folder with a different number of folders, and a slightly different naming convention than what your tool expected.

You can't rely on your database to get information from these paths because neither paths have actually been published yet - just rendered to disk.

In Ways, the same situation can be solved by just writing a new plugin

```
plugins:
    linux_root:
        hierarchy: job
        mapping: /jobs
        path: true
    element:
        hierarchy: '{root}/shot/element'
        mapping: '{root}/{JOB}/{SCENE}/{SHOT}/elements'
        uses:
            - job
    sequence_bit:
        hierarchy: '{root}/rendered/sequence'
        mapping: '{root}/{NAME}/{VERSION}/{LAYER}/{SEQUENCE_NAME}'
        uses:
            - job/shot/element
    houdini_rendered_plugin:
        hierarchy: '{root}/rendered/sequence/houdini'
        mapping: '{root}/plates/houdini/{NAME}_{VERSION}/{VERSION}/{LAYER}/file_
↪sequence.####.tif'
        uses:
            - job/shot/element
```

```
houdini_rendered_plugin:
    hierarchy: '{root}'/rendered/sequence/houdini'
    mapping: '{root}/plates/houdini/{NAME}_{VERSION}/{VERSION}/{LAYER}/file_sequence.#
↪###.tif'
    uses:
        - job/shot/element
```

Adding houdini_rendered_plugin was all we needed to do. Now we can publish those paths without changing anything else.

```
path1 = "/jobs/fooJob/fooScene/sh01/elements/frame_Render/v001/beauty/file_sequence.##
↪##.tif"
path2 = "/jobs/{JOB}/{SCENE}/{SHOT}/elements/plates/houdini/frame_render_001/v1/rgba/
↪file_sequence.####.tif"
asset1 = ways.api.get_asset(path1)
asset2 = ways.api.get_asset(path2)

asset1.actions.publish()
asset2.actions.publish()
```

Note: When no context is given to "get_asset", Ways will guess the "best" possible Context for the information you give it.

If the information was a string and it matches a Context's mapping, this guess will always be correct.

There's more information about this in *mapping* and *Auto-find using mapping*.

Both plugins, "sequence_bit" and "houdini_rendered_plugin", share the same hierarchy - "job/shot/element". "job/shot/element" has a "publish" Action defined for it so our new hierarchy in "job/shot/element/rendered/sequence/houdini" can also reuse the same Action.

1.2.7 Split Deployment

Sometimes even the perfect tool must change. Maybe the client has a special job that needs to ingest filepaths from a different location.

So normally, your tool would point to one filepath, "/some/filepath/here" but for one specific setup, it needs to "/some/other/path/here". And both setups are in use at the same time.

Depending on your environment's setup, this may not be trivial to do. Thankfully though, it is trivial to do in Ways, by using something that Ways calls "plugin assignment". It's an advanced feature that isn't often used.

A couple sections in another page, *Using Assignments* is dedicated to show how to do this so, if you're curious how it works, check it out there.

1.3 Getting Started

This section assumes that you've read through the *API Summary* Page and that you're ready to start.

First, create a new YAML file, call it whatever you'd like. For this example, it'll be called "plugin_sheet.yml". Add the path to this file to your WAYS_DESCRIPTORS environment variable.

1.3.1 Setting your environment variable

tcsh/csh

```
setenv WAYS_DESCRIPTORS /some/path/to/a/plugin_sheet.yml
```

bash

```
export WAYS_DESCRIPTORS=/some/path/to/a/plugin_sheet.yml
```

Your specific command will depend on your terminal/setup.

1.3.2 Writing your your first Plugin Sheet

Now that plugin_sheet.yml exists, lets add a "Hello World!" plugin to it.

```
> cat /some/path/to/a/plugin_sheet.yml
plugins:
    foo_plugin:
        hierarchy: 'some/context'
```

At this point, we've made our first plugin in our first Plugin Sheet. Now you can open a Python interpreter or import it into another file and use it as a `ways.api.Context`.

1.3.3 Create your new Context

```
import ways.api
context = ways.api.get_context('some/context')
```

If context is not None, congratulations, Ways is ready to use.

1.3.4 Adding features to Context objects

From here, we can add data to the Context

```
> cat /some/path/to/a/plugin_sheet.yml
plugins:
    foo_plugin:
        hierarchy: 'some/context'
        data:
            some:
                arbitrary:
                    - info1
                    - info2
                    - 3
                    - bar
```

```
import ways.api
context = ways.api.get_context('some/context')
context.data['some']['arbitrary']
# Result: ['info1', 'info2', 3, 'bar']
```

Here we can add data to the plugin and access it later, though a Context. Context objects persist everywhere as long as you call them with the same hierarchy.

```
context = ways.api.get_context('some/context')
print(id(context))
context.data['some']['arbitrary'].append('bar2')

def some_function():
    a_new_context = ways.api.get_context('some/context')
    print(id(a_new_context))
    print(a_new_context.data['some']['arbitrary'])
    # Result: ['info1', 'info2', 3, 'bar', 'bar2']
```

In the above example, we have a Context that initializes with some metadata, we add to the metadata, and then we call the same hierarchy in another function. The Context in "some_function" already has the data that was appended, earlier. You're allowed to write anything in a Context's data.

There's a lot more to how Plugin objects are defined. Including Context inheritance, relative plugins, and OS-aware plugins. To know more, Check out *Plugin Basics* and *Advanced Plugin Topics*.

1.3.5 Asset Objects

We have a generic description of a path on disk "/some/{JOB}/and/folders" so now we'll extend it using an Asset object.

If Context objects are branches on a tree, think of Asset objects as the leaves. Meaning, Context objects describe a range of information and Asset objects are specific points along that range. There can only be 1 of any Context but there could be any number of Asset objects.

Creating an Asset object is more or less the same as creating a Context. The main difference is that any part of a Context's mapping that is an unfilled Token (in our above example "{JOB}" is unfilled), we need to define it.

```
> cat /some/path/to/a/plugin_sheet.yml
plugins:
    job:
        hierarchy: 'some/context'
        mapping: /jobs/{JOB}/here
```

```
# All 3 of these syntaxes create the same Asset object
asset1 = ways.api.get_asset((('JOB', 'foo'), ), 'some/context')
asset2 = ways.api.get_asset({'JOB': 'foo'}, 'some/context')
asset3 = ways.api.get_asset('/jobs/foo/here', 'some/context')
print(asset1.get_str())
# Result: '/jobs/foo/here'
print(asset1.get_value('JOB'))
# Result: 'foo'
```

Asset objects act like dictionaries that have some data and the Context is what grounds that dictionary in something real (i.e. a filesystem or a database). Asset objects have a small list of features that you'll learn in other sections, like token validation (checking if tokens are optional or not), Context-expansion, recursive value parsing, and API hooks so that you can swap Asset objects for classes that you may have already written. To find out more about that, check out *Asset Object Tricks*.

1.3.6 Context Actions

Great - we have a Context and Asset object. You may have noticed though that both classes have very few methods. Ways tries to not assume how you'll use Context and Asset objects and instead lets you to extend the object's interfaces at runtime, using Actions.

To create an Action for our original example, create a new file name anything - we'll call ours action.py. Add the path to action.py into the WAYS_PLUGINS environment variable.

Now just add a new class in action.py, have it inherit from ways.api.Action, and implement two methods.

plugin_sheet.yml

```
plugins:
    foo_plugin:
        hierarchy: 'some/context'
```

action.py

```
import ways.api

class SomeAction(ways.api.Action):

    '''A subclass that will automatically be registered by Ways.

    The name of the class (SomeAction) can be anything but the name
    property must be correct. Also, get_hierarchy must match the Context
    hierarchy that this action will apply to.

    '''

    name = 'create'
    @classmethod
    def get_hierarchy(cls):
        return 'some/context'

    def __call__(self, obj):
        '''Do something.'''
        return ['/library', 'library/grades', 'comp', 'anim']
```

Note: __call__ takes at least one arg - the Context or Asset that called the Action. Ways will pass the caller object to this variable before any of the user's args/kwargs.

To use the Action that was just created, call it from a Context or Asset.

```
context = ways.api.get_context('some/context')
context.actions.create()
# Results: ['/library', 'library/grades', 'comp', 'anim']
```

That's all there is to it. If you don't want to write an Action subclass, you can also use a regular function and register it.

```
def some_action(obj):
    return ['/library', 'library/grades', 'comp', 'anim']

context = ways.api.get_context('some/context')
ways.api.add_action(some_action, context='some/context')
context.actions.some_action()

# If you don't want to use the name of the function, you can give the action
# a name
#
ways.api.add_action(some_action, 'custom_name', context='some/context')

context.actions.custom_name()
# Result: ['/library', 'library/grades', 'comp', 'anim']
```

It doesn't matter what the order of your objects are defined. Actions that are defined before Context/Asset objects will work fine too. All that matters is that both exist by the time you call the Action from a Context.

1.3.7 Context and Asset Actions

We've been using Context.actions this whole time but Asset objects have an "actions" property, too.

Actions called from an Asset object behave the same a Context objects. The only difference is that the first arg that get's passed to the Actions object will be the instance of Asset that called it, not the Context.

If we want to call get_info from an Asset instance and pass it the Context, we still can.

```
asset = ways.api.get_asset({'JOB': 'something'}, context='some/context')

# Using the Context object
context = ways.api.get_context('some/context')
context.actions.get_info()  # get_info will pass 'context'

# Using the Context located in the Asset object
asset.context.actions.get_info()  # get_info will pass 'asset.Context'
```

```
# This is still the preferred way, most of the time
asset.actions.get_info()  # get_info will pass 'asset'
```

The most powerful way to chain Actions together is to have Action objects return other Context/Asset/Action objects. Actions have very few rules and can be formatted to your needs easily.

Now that you've gone through the basics, make sure to read through *Plugin Basics* to get familiar with Ways and *Common Patterns And Best Practices* to get an idea of how you should be formatting your code.

CHAPTER 2

API Details

2.1 Descriptors

This section assumes that you've at least read through the *Descriptor Objects* section.

2.1.1 Basic setup

When Ways initializes, it gathers all of the Descriptor objects it can find, using your environment.

The most basic Descriptor is usually a path to a folder or file. All descriptors added to the WAYS_DESCRIPTORS environment variable are loaded into Ways, by default.

Example:

```
export WAYS_DESCRIPTORS=/tmp/to/plugins/folder:/tmp/to/plugin.yml:/tmp/to/plugin.
↪json:/tmp/to/plugin.py
```

The above example is 4 different ways to load a Plugin Sheet or file. In each case, Ways will convert those paths to Descriptor objects and then load the plugins that those objects find.

2.1.2 Descriptors Under The Hood

Custom Descriptors still use the WAYS_DESCRIPTORS environment variable the same as a path-based Descriptor like we defined earlier but the string is a standard URL encoding instead of a place on-disk. For those who don't know, URL encoding is just way to serialize a dictionary into a string, similar to JSON or YAML.

```
{
    "key1": "value1",
    "key2": "value2"
}
```

Converts to

```
key1=value1&key2=value2
```

Here's a real example of what a Ways custom Descriptor looks like:

```
path=%2Fsome%2Fpath%2Fon%2Fdisk&class_type=ways.api.GitLocalDescriptor&items=plugins
```

This example string describes a local git repository.

when our cache unpacks this descriptor string, the result is a dict

```
descriptor_info = {
    'create_using': 'ways.api.GitLocalDescriptor',
    'uuid': 'some_unique_string-we-can_search_for-later',
    'path': '/some/path/on/disk',
    'items': ['plugins'],
}
```

"create_using" and "uuid" are the only reserved key in our dict for all custom Descriptors. Ways uses "create_using" to import the Descriptor object. "uuid" is used to get the Descriptor later. For example, if a Descriptor failed to load and you wanted to get its load results to find out why it failed to load, you'd use the value in "uuid" to do it.

Excluding "create_using" and "uuid", all other key/value pairs are passed to the Descriptor, directly. It's worth noting that create_using can be any callable Python object. It could be a function or a class, for example.

Knowing what you now know about Descriptors, the previous example with the 4 different ways to load Descriptors could technically be rewritten as URL strings.

```
/tmp/to/plugins/folder
items=%2Ftmp%2Fto%2Fplugins%2Ffolder&create_using=ways.api.FolderDescriptor
```

```
/tmp/to/plugin.yml
items=%2Ftmp%2Fto%2Fplugin.yml&create_using=ways.api.FileDescriptor
```

```
/tmp/to/plugin.json
items=%2Ftmp%2Fto%2Fplugin.json&create_using=ways.api.FileDescriptor
```

```
/tmp/to/plugin.py
items=%2Ftmp%2Fto%2Fplugin.py&create_using=ways.api.FileDescriptor
```

```
export WAYS_DESCRIPTORS=/path/to/plugins/folder:/path/to/plugin.yml:/path/to/plugin.
↪json:/path/to/plugin.py:/path/to/plugin/folder
export WAYS_DESCRIPTORS=items=%2Ftmp%2Fto%2Fplugins%2Ffolder&create_using=ways.api.
↪FolderDescriptor:items=%2Ftmp%2Fto%2Fplugin.yml&create_using=ways.api.
↪FileDescriptor:items=%2Ftmp%2Fto%2Fplugin.json&create_using=ways.api.
↪FileDescriptor:items=%2Ftmp%2Fto%2Fplugin.py&create_using=ways.api.FileDescriptor
```

It should be pretty obvious that the first line is easier to understand than the URL-encoding method. But the URL-encoding method is useful for whenever you need a custom Descriptor load behavior.

2.1.3 Database Descriptors

It was hinted at in the previous section that Ways supports reading Git repositories directly, instead of using the filesystem. If storing Plugin Sheet files locally isn't an option, reading from server is an alternative.

Any callable object can be a Descriptor

Any object that is callable (functions, methods, classes) can be a Descriptor.

```
path=%2Ftmp%2Fpath%2Fsome_module.py&create_using=some_module.some_function&
↪items=plugins
```

/tmp/path/some_module.py

```
def some_function(*args, **kwargs):
    return [ways.api.Plugin()]
```

Ways will try to assume that the Descriptor object passed is a class and run "get_plugins". If that fails, Ways tries to call the object, directly, as though it was a function.

Custom Descriptors

If you have your own I/O requirements that Ways doesn't handle out of the box, you can write your own Descriptor and use it.

Descriptors requires two methods to be supported by Ways: One method to get Plugin objects and another method to display those objects's information.

Ways expects and looks for a method named "get_plugins". If the Descriptor object doesn't have a "get_plugins" method, then it must be callable. Either way, the method's return should be a list of Plugin Objects. Every plugin found will be given the assignment "master" by default unless you specify otherwise.

Here is an example of a custom Descriptor.

```
class CustomDescriptor(object):
    def get_plugins(self):
        return [CustomPlugin()]
```

In this example, the Descriptor will always return one plugin, CustomPlugin(). This Plugin object will be given the assignment of "master" (or whatever ways.api.DEFAULT_ASSIGNMENT is). If you need the Plugin to go to a different assignment, just specify it in get_plugins.

```
class CustomDescriptor(object):
    def get_plugins(self):
        return [(CustomPlugin(), 'foo')]
```

The method used to display objects's information is optional but highly recommended because it's needed for some of Ways's more advanced features. It should be called "get_plugin_info" and return a dict with any data about the Plugins that can't be stored on the Plugins, themselves. For example, the default implementation of Ways looks for a file called ".ways_plugin_info" in directories on or above wherever Plugin Sheets are loaded.

```
import ways.api

class CustomPlugin(ways.api.Plugin):

    data = {'data': True}

    @classmethod
    def get_hierarchy(cls):
        return ('something', 'here')
```

```
class CustomDescriptor(object):
    def get_plugins(self):
        return [(CustomPlugin(), 'master')]

    def get_plugin_info(self):
        return {'assignment': 'master', 'foo': 'bar'}
```

The last things to do are to make sure that CustomDescriptor is importable on the PYTHONPATH and it can be used like any other Descriptor.

Custom descriptors can be called using URL syntax using WAYS_DESCRIPTORS or by including a python file in WAYS_PLUGINS and registering the descriptor, directly. Either method will work.

```
info = {
    'create_using': 'ways.tests.test_documentation.CustomDescriptor',
}
ways.api.add_descriptor(info)
context = ways.api.get_context('something/here')
print(context.data['data'])
# Result: {'data': True}
```

2.2 Plugin Basics

Plugin Sheets are the backbone of Ways. They define many plugins in very few lines and drive how Ways will parse your objects.

Because Plugin Sheets have a finite list of keys that it can use, it's important to know what each of them are called and what each of them do.

2.2.1 All Plugin Sheet Keys

This is a "Hello World" Plugin Sheet. It's the absolute minimum information that every Plugin Sheet has. The only things that are required is a single value under "plugins" and that value must have a "hierarchy" defined.

Note: For reference, this is a YAML file. YAML is used in this example because it's pretty easy to read and follow but Ways also supports JSON and Python files.

```
plugins:
    some_plugin:
        hierarchy: example
```

Here is an example of a very complicated Plugin Sheet. Every key that Ways uses is in this file. This is a bit like getting thrown into the deep-end of a pool but don't worry if not everything makes sense immediately. It'll all be explained in this page and in others.

```
globals:
    assignment: an_assignment_to_every_plugin
plugins:
    some_plugin:
        hierarchy: example
        uuid: something_unique
    this_can_be_called_anything:
        hierarchy: example/hierarchy
        mapping: "/jobs/{JOB}"
        uuid: another_unique_uuid
        platforms:
            - linux
        path: true

    window_jobs_plugin:
        hierarchy: example/hierarchy
        mapping: "C:\\Users\\{USER}\\jobs\\{JOB}"
        mapping_details:
            USER:
                parse:
                    regex: \w+
        platforms:
            - windows
        uuid: windows_job
        path: true

    jobs_details:
        hierarchy: example/hierarchy
        mapping_details:
            JOB:
                mapping: '{JOB_NAME}_{JOB_ID}'
            JOB_NAME:
                mapping: '{JOB_NAME_PREFIX}_{JOB_NAME_SUFFIX}'
            JOB_NAME_PREFIX:
                parse:
                    regex: '\w+'
            JOB_NAME_SUFFIX:
                parse:
                    regex: 'thing-\w+'
            JOB_ID:
                parse:
                    regex: '\d{3}'
        uuid: something_unique

    yet_another_plugin:
        hierarchy: example/tree
        mapping: /tree
        uuid: does_not_matter_what_it_is
        path: true

    config_plugin:
        hierarchy: "{root}/config"
        mapping: "{root}/configuration"
        uses:
            - example/hierarchy
            - example/tree
        uuid: as_Long_as_It_is_Different

    some_assigned_plugin:
        assignment: different_assignment
        hierarchy: something
        data:
            important_information: here
        uuid: boo_Did_I_scare_you?
```

Clearly there is a big difference between a "Hello World" Plugin Sheet and this one. The good news is, everything in this example optional and you may not need to ever use it all.

Feel free to use this page as a reference while writing Plugin Sheets.

2.2.2 Required Keys

plugins

This is the only required, top-level key. It is a dictionary that contains all of the plugins inside the Plugin Sheet. As long as its items are valid dict keys, anything can be used as a plugin key though it's recommended to use strings, since they're easy to read. Example plugin keys from above are "some_plugin", "this_can_be_called_anything", "job_details", and all of the other defined plugins.

hierarchy

This is the only required key at the plugin-level. The value that's defined for hierarchy must be a string, separated by "/"s (even if you're using Windows). The hierarchy is used to create objects so it's important that it is named sensibly.

2.2.3 Optional Keys

globals

This key lives on the same level as the "plugins" key and is a quick way to add information to every plugin in the file.

In the above example, "assignment" was added to globals. That key/value is added to every plugin in the file, unless the plugin overrides it. In the above example, every plugin will have the assignment "an_assignment_to_every_plugin" except for some_assigned_plugin, which will have an assignment of "different_assignment".

mapping

mapping is just a string that describes a plugin. The complex example above treats its mapping like it's a filepath but mapping doesn't have to be a file or folder. It can be anything. For example, mapping can be used to reference a database, too.

When you begin to use Asset objects (*ways.api.Asset*), the mapping becomes crucial for "auto-finding" Context (*ways.api.Context*) objects.

```
mapping = '/jobs/job_part_something'

explicit_asset = ways.api.get_asset(mapping, context='example/hierarchy')
autofound_asset = ways.api.get_asset(mapping)
explicit_context == autofound_context
# Result: True
```

the right Context should be, mapping is something necessary. The mapping and uuid keys are always a good idea to define.

For practical examples on using mapping, see *Common Patterns And Best Practices*.

mapping_details

Anything in “{}” inside of a mapping is called a “Token”. Above, “/jobs/{JOB}” has a “JOB” Token and “C:\Users\{USER}\jobs\{JOB}” has “USER” and “JOB” Tokens.

Tokens look like a Python format but have a set of features specific to Ways.

For one thing, Tokens can represent environment variables or parse-engines like regex and glob.

```
os.environ['JOB'] = 'job_thing-something_123'

context = ways.api.get_context('example/hierarchy')
context.get_str(resolve_with=('env', 'regex'))
# Result on Windows: "C:\\Users\\My_Username\\jobs\\thing-something"
# Result on Linux/Mac: "/jobs/thing-something"

# Both calls, 'regex' and ('regex', ), do the same thing
context.get_str(resolve_with='regex')
context.get_str(resolve_with=('regex', ))
# Result on Windows: "C:\\Users\\\w+\\jobs\\w+_thing-\w+_\d{3}"
# Result on Linux/Mac: "/jobs/\w+_thing-\w+_\d{3}"
```

If you’ve read the *Why use Ways* link, this example will look familiar.

Immediately, you should take note of a few things. resolve_with=(‘env’, ‘regex’) will try to fill in the mapping with environment variables first, and then fall back to regex if it can’t. Changing resolve_with=’regex’, makes get_str ignore any environment variables and grab only regex values.

The second important thing to note is that the regex for “JOB”, which is “w+_d{3}”, wasn’t actually defined in JOB. It was defined in Subtokens, JOB_NAME_PREFIX and JOB_NAME_SUFFIX and JOB_ID. Ways composed that regex value for JOB using its Subtokens. Like the name implies, a Subtoken is a Token that is nested inside of another Token.

In docstrings, we refer to this as a “Child-Search”. Ways also has a “Parent-Search” which is exactly like “Child-Search” but instead of searching for values down, it looks up at a Subtoken’s parents. Both Child-Search and Parent-Search are recursive.

Search methods like Parent/Child Search matter once you start getting into the deeper parts of Ways, such as Asset objects. For now, just know that it exists.

```
mapping = '/jobs/job_thing-something_123'
asset = ways.api.get_asset(mapping, context='example/context')
asset.get_value('JOB_NAME_SUFFIX')
# Result: 'thing-something'
```

By the way, get_value can work on its own, *with or without regex*. Regex is good to have but is not required.

Just like how mapping is used to find Contexts automatically when none is given, mapping_details is used to find values for mapping automatically when pieces are missing.

uuid

```
plugins:
    something:
        hierarchy: foo/bar
        uuid: some_string_to_describe_this_plugin
```

This is just a string that Ways will use to refer to your plugin. It can be an actual UUID (http://docs.python.org/3/library/uuid.html) or anything else, as long as it’s unique.

If you find yourself needing to troubleshoot a Context or Asset, some of the tools that Ways has will require a UUID.

There's more information about this in *Common Patterns And Best Practices* and *Troubleshooting Ways*.

data

data is a regular dictionary that gets loaded onto the Context once it is first created. It's mostly just a place to store metadata onto and retrieve later. You can also modify and add to data like a regular dictionary in a live Python session to an extent.

There's two things you'll want to know about data before you use it.

The first is that there's a separation between "loaded" values and "user" values. Loaded values come for the the plugin files that are registered to Ways. These keys/values cannot be removed. Then there are user values, which are keys that you can edit, add, and remove freely. You can change values from the loaded plugin data but you cannot delete it.

If you ever need to go back to a Context's initial data, just call Context.revert().

platforms

platforms refers to the operating system(s) that a plugin is allowed to run on.

Ways has two environment variables related to the "platforms" key, WAYS_PLATFORM and WAYS_PLATFORMS.

WAYS_PLATFORM

Every plugin has a set of platforms that it's allowed to run on. If one of the platforms in the plugin matches the WAYS_PLATFORM environment variable, Ways will use it. If WAYS_PLATFORM isn't defined, Ways will just use the computer's OS, instead.

```yaml
plugins:
    explicit_star_platform:
        hierarchy: foo
        platforms:
            - '*'
    implicit_star_platform:
        hierarchy: bar
    some_platforms:
        hierarchy: fizz
        platforms:
            - linux
            - darwin
```

If "*" is a platform on a plugin, then it is automatically assumed that the plugin works on everything. Any plugin with no platforms defined, like "implicit_star_platform" will get "*" by default.

WAYS_PLATFORMS is the list of platforms that Ways knows about. It can be any string that you'd like, separated by your OS path separator (":" in Linux, ";" in Windows). If WAYS_PLATFORMS isn't defined, a default set of platforms if given instead.

TODO : Write a very concise platform example

There's a really good example of how to use platforms in *Designing For Cross-Platform Use* if you'd like to see another example.

uses

The difference between an absolute plugin and a relative plugin is whether or not "uses" is defined. There's a lot to talk about when it comes to absolute vs. relative plugins and it is explained on other pages so, in summary, for now relative plugins can be explained as "plugins that create plugins". They're a huge time saver and make Plugin Sheets easier to understand. For more information on how they're built, check out *Advanced Plugin Topics* for details.

assignment

All plugins have assignments. If no assignment is given to a plugin when it is first created, the plugin is given a default "master" assignment.

The assignment key is one of the most important keys because it can drastically change how Ways runs in very little lines. In a single sentence, assignment has the flexibility of "platforms" and the re-usability of "uses". For more information on how to use them, check out *Advanced Plugin Topics* for details.

path

If you are developing a hierarchy that represents a filepath and you need to support more than one type of OS (like Linux and Windows), it's best to set this option to True.

On Linux, setting path forces "" in a mapping to "/". On Windows, it changes "/" to "".

Ways will use the OS you've defined in the WAYS_PLATFORM environment variable. If that environment variable is not set, Ways will use your system OS.

The path key exists because path-related plugins are difficult to write for more than one OS at a time. Take the next example. If we got the mapping for "foo/bar", with the Plugin Sheet below, we get an undesired result on Windows.

```
plugins:
    path_plugin:
        hierarchy: foo
        mapping: '/jobs/{JOB}'
        platforms:
            - linux
    windows_path_root:
        hierarchy: foo
        mapping: 'Z:\jobs\{JOB}'
        platforms:
            - windows

    relative_plugin:
        hierarchy: '{root}/bar'
        mapping: '{root}/shots'
        uses:
            - foo
```

```
context = ways.api.get_context('foo/bar')
context.get_mapping()
# Result on Linux: '/jobs/{JOB}/shots'
# Result on Windows: 'Z:\jobs\{JOB}/shots'
```

The result on Windows is mixes "" and "/" because the relative plugin used "/". If we include path: true, this isn't a problem.

```
plugins:
    path_plugin:
        hierarchy: foo
        mapping: '/jobs/{JOB}'
        platforms:
            - linux
    windows_path_root:
        hierarchy: foo
        mapping: 'Z:\jobs\{JOB}'
        platforms:
            - windows
    plugin_that_appends_path:
        hierarchy: foo
        path: true

    relative_plugin:
        hierarchy: '{root}/bar'
        mapping: '{root}/shots'
        uses:
            - foo
```

```
context = ways.api.get_context('foo/bar')
context.get_mapping()
# Result on Linux: '/jobs/{JOB}/shots'
# Result on Windows: 'Z:\jobs\{JOB}\shots'
```

The "foo" hierarchy is set as a path so its child hierarchy, "foo/bar" also becomes a path. Now things work as we expect.

What Now?

Now that you know the basics of each key, head over to *Advanced Plugin Topics* or *Common Patterns And Best Practices* to see examples of these keys in examples.

2.3 Advanced Plugin Topics

2.3.1 Relative Plugins

One of the most fundamental ideas about Plugin Sheets is that there are two types of plugins, relative plugins and absolute plugins.

The most bare-minimum absolute plugin looks like this:

```
plugins:
    absolute_plugin:
        hierarchy: fizz/buzz
```

The plugin contains just one item, "hierarchy", which is the position of the Plugin for when it gets built into a Context.

A bare-minimum relative plugin looks like this

```
plugins:
    absolute_plugin:
        hierarchy: fizz
    relative_plugin:
        hierarchy: '{root}/buzz'
        uses:
            - fizz
```

A relative plugin can also refer to another relative plugin recursively, as long as the end of that chain of plugins is an absolute plugin.

Calling a plugin "relative" is a bit of a inaccurate. Relative plugins are not single plugins - they're a group of plugins. Each hierarchy listed under "uses", will create a separate Plugin object.

Note: {root} is only supported on a plugin's hierarchy and mapping but it is also not required. If no {root} is given, Ways will just append the relative plugin's mapping and hierarchy to its parent. If you do provide {root} though, you get to define different places for the parent's data to be inserted, like this: "parent/{root}/library/{root}/hierarchy".

"uses" has a couple details that are important to know before starting.

1. uses should never give a relative plugin its own hierarchy. For example, these setups are invalid:

```
plugins:
    relative:
        mapping: something
        hierarchy: some/place
        uses:
            - some/place
```

```
plugins:
    absolute:
        mapping: whatever
        hierarchy: foo
    relative:
        mapping: "{root}/something"
        hierarchy: "{foo}/bar"
        uses:
            - foo/bar
```

2. Relative plugins can be chained together, as long as one of the plugins is tied to an absolute plugin.

```
plugins:
    absolute_plugin:
        hierarchy: fizz
    relative_plugin1:
        hierarchy: '{root}/buzz'
        uses:
            - fizz
    relative_plugin2:
        hierarchy: '{root}/foo'
        uses:
            - fizz/buzz
```

The initial setup for relative plugins is a bit verbose but has its advantages. The main advantage is re-useability.

Here is an example of how absolute plugins and relative plugins differ.

Relative

```
plugins:
    absolute_plugin:
        hierarchy: fizz
        mapping: bar

    relative_plugin1:
        hierarchy: '{root}/buzz'
        mapping: '{root}/something'
        uses:
            - fizz

    absolute_plugin2:
        hierarchy: '{root}/pop'
        mapping: '{root}/another/thing'
        uses:
            - fizz/buzz

    absolute_plugin3:
        hierarchy: '{root}/fizz'
        mapping: '{root}/sets'
        uses:
            - fizz/buzz/pop

    library:
        hierarchy: '{root}/library'
        mapping: '{root}/library'
        uses:
            - fizz
            - fizz/buzz
            - fizz/buzz/pop
            - fizz/buzz/pop/fizz
```

Absolute

```
plugins:
    absolute_plugin:
        hierarchy: fizz
        mapping: bar

    absolute_plugin1:
        hierarchy: fizz/buzz
        mapping: bar/something

    absolute_plugin1_library:
        hierarchy: fizz/buzz/library
        mapping: bar/something/library

    absolute_plugin2:
        hierarchy: fizz/buzz/pop
        mapping: bar/something/another/
↪thing

    absolute_plugin2_library:
        hierarchy: fizz/buzz/pop/library
        mapping: bar/something/another/
↪thing/library

    absolute_plugin3:
        hierarchy: fizz/buzz/pop/fizz
        mapping: bar/something/another/
↪thing/sets

    absolute_plugin3_library:
        hierarchy: fizz/buzz/pop/fizz/
↪library
        mapping: bar/something/another/
↪thing/sets/library
```

Both examples create the same exact Plugins.

So to compare the two examples - the relative plugin example took more lines to create the absolute plugin version. If this example were longer however, the relative plugin version would come out shorter because each line in “uses” is 3 lines in the absolute version.

Also, if we needed to change something in “library”, we only need to change one plugin in the relative system, whereas in an absolute system, you would need to change it in 3 places.

Note: When Ways loads Plugins, all Plugins are “resolved” into absolute Plugin objects.

2.3.2 Designing For Cross-Platform Use

If you’re using Ways to build Context objects for your filesystem, you may have to consider supporting multiple operating systems.

Say you have two paths that represent the same place on-disk in Windows and in Linux: /jobs/someJobName_123/library and Windows: \NETWORK\jobs\someJobName_123\library.

You might be tempted to write your plugins like this:

```
plugins:
    linux:
        mapping: /jobs
        hierarchy: job
    windows:
        mapping: \\NETWORK\jobs\someJobName_123\library
        hierarchy: job
    linux_library:
        mapping: /jobs/someJobName_123/library
        hierarchy: job/library
    windows_library:
        mapping: \\NETWORK\jobs\someJobName_123\library
        hierarchy: job/library
    linux_library_reference:
        mapping: /jobs/someJobName_123/library/reference
        hierarchy: job/library/reference
    windows_library_reference:
        mapping: \\NETWORK\jobs\someJobName_123\library\reference
        hierarchy: job/library/reference
```

This works but you wanted to keep data consistent across both plugins, you'd be forced to write separate plugins for each OS and each feature.

To make the process easier, just use relative plugins

```
plugins:
    job_root_linux:
        hierarchy: job
        mapping: /jobs
        platforms:
            - linux

    job_root_windows:
        hierarchy: job
        mapping: \\NETWORK\jobs
        platforms:
            - windows

    library:
        hierarchy: '{root}/library'
        mapping: '{root}/someJobName_123/library'
        uses:
            - job

    reference:
        hierarchy: '{root}/reference'
        mapping: '{root}/reference'
        uses:
            - job/library
```

Imagine a scenario where you have to maintain 100 separate plugins for Windows, Mac, and Linux. If it were written using absolute plugins only, that'd mean writing 300 plugins, total. And if you needed to change any plugin, you'd have to change it once for each OS. But if we write 3 absolute plugins, each with their own platform, we can write 99 relative plugins that just append to that root OS plugin. Ways will pick whichever root matches the system OS or what is defined in the WAYS_PLATFORM environment variable and then append all other 99 plugins onto it.

Note: If you're designing plugins for cross-platform use and you're dealing with filepaths, it's best to write "path: true" in your hierarchies. That way, the path separator will always be correct. Examples of this are in *path* and in *Common Patterns And Best Practices*.

2.3.3 Appending To Plugins

Say for example you have a plugin in another file that you want to add to. You have two options to do this, an absolute append or a relative append.

You can do this using a relative plugin, but isn't generally a good idea because its syntax is harder to follow

```
plugins:
    some_plugin:
        hierarchy: foo/bar
        mapping: something
    append_plugin:
        hierarchy: ''
        data:
            some_data: 8
        uses:
            - foo/bar
```

Appending with an absolute plugin is much simpler.

```
plugins:
    some_plugin:
        hierarchy: foo/bar
        mapping: something
    append_plugin:
        hierarchy: foo/bar
        data:
            some_data: 8
```

But if you need to append information to more than one plugin at once, relative plugins are very useful.

```
plugins:
    some_plugin:
        hierarchy: foo/bar
        mapping: something
    another_plugin:
        hierarchy: another
    another_plugin:
        hierarchy: another/tree
    append_plugin:
        hierarchy: ''
        data:
            some_data: 8
        uses:
            - foo/bar
            - another
            - another/tree
```

So in conclusion, absolute and relative plugins both have their pros and cons. Pick the right one for the right job.

Other than plugin platforms and relative plugins, there's another way to affect the discovery and runtime of plugins in Ways called "assignments".

2.3.4 Using Assignments

Whenever a Plugin is defined, its hierarchy is defined and if no assignment is given, ways.DEFAULT_ASSIGNMENT is used, instead.

Ways assignments allow users to change the way plugins resolve at runtime.

First lets explain the syntax of assignments and then explain how this works in a live environment.

There are 3 ways to define assignments to a plugin. Each one is a matter of convenience/preference and is no better than the other.

Assigning To Multiple Plugin Sheets

With the default Ways Descriptor classes, if you have a file called ".waypoint_plugin_info" in the same directory or above a Plugin Sheet, any assignment listed is used.

".waypoint_plugin_info" can be JSON or YAML.

Examples:

```
>>> cat .waypoint_plugin_info.json
>>> {
>>>     "assignment": master,
>>>     "recursive": false
>>> }
```

```
>>> cat .waypoint_plugin_info.yml
>>> assignment: master
>>> recursive: false
```

Note: "recursive" defines if we will search for Ways Plugin Sheets in subfolders. For more information, *seealso environment_setup.rst*

The assignment in this file will apply to all plugins in all Plugins Sheets at the same directory or below the ".waypoint_plugin_info" file.

Assigning To A Plugin Sheet

You can add an assignment to every plugin in a Plugin Sheet, using "globals"

```
globals:
    assignment: bar
plugins:
    some_plugin:
        hierarchy: some/hierarchy
    another_plugin:
        hierarchy: another/hierarchy
```

All plugins listed now have "job" assigned to them. Using "globals" takes priority over any assignment in a ".waypoint_plugin_info" file.

Assigning To A Plugin

If an assignment is directly applied to a plugin, then it is used over any other assignment method.

```
plugins:
    another_plugin:
        hierarchy: another/hierarchy
        assignment: job
```

2.3.5 Applied Assignments - Live Environments

Whenever you call a Context, you must give a hierarchy and an assignment. If no assignment is given, Ways "searches" for plugins in every assignment that it knows about, defined in the WAYS_PRIORITY environment variable.

```
export WAYS_PRIORITY=master:shot:job
```

In the above example, "master" plugins are loaded first, then "job" plugins, and then "shot" plugins.

To take advantage of this in a live environment, here is a short example.

master.yml

```
plugins:
    job:
        hierarchy: job
        mapping: '/jobs/{JOB}'
    shot:
        hierarchy: '{root}/shot'
        mapping: '{root}/{SCENE}/{SHOT}'
        uses:
            - job
    plates:
        hierarchy: '{root}/plates'
        mapping: '{root}/library/graded/plates'
        uses:
            - job/shot
    client_plates:
        hierarchy: '{root}/client'
        mapping: '{root}/clientinfo'
        uses:
            - job/shot/plates
    compositing:
        hierarchy: '{root}/comp'
        mapping: '{root}/compwork'
        uses:
            - job/shot/plates
```

Here, we didn't define an assignment and we have no ".waypoint_plugin_info.(yml|json)" file, so ways.DEFAULT_ASSIGNMENT (master) is given to every Plugin.

Now define the WAYS_PRIORITY

sh/bash

```
export WAYS_PRIORITY=master:job
```

csh/tcsh

```
setenv WAYS_PRIORITY master:job
```

Add a folder or file location to the WAYS_DESCRIPTORS environment variable where we're going to look for "job-specific" Plugin Sheets.

```
export WAYS_DESCRIPTORS=/path/to/master.yml:/path/to/job/plugins
```

The last step is to add a 'job'-assigned Plugin Sheet to the /path/to/job/plugins folder.

jobber.yml

```
globals:
    assignment: job
plugins:
    job_plugin:
        hierarchy: '{root}/plates'
        mapping: '{root}/archive/plates'
        uses:
            - job/shot
```

Now let's see this in a live Python environment

```
# Both get_context versions do the same thing, because assignment='' by default
context = ways.api.get_context('job/shot/plates/client', assignment='')
context = ways.api.get_context('job/shot/plates/client')
context.get_mapping()
# Result: "/jobs/{JOB}/{SCENE}/{SHOT}/archive/plates/clientinfo"
```

Before adding jobber.yml to our system, the mapping for "job.shot/plates/client" was "/jobs/{JOB}/{SCENE}/{SHOT}/library/graded/plates/clientinfo".

Now, it's "/jobs/{JOB}/{SCENE}/{SHOT}/archive/plates/clientinfo".

This works because the "job_plugin" key in jobber.yml matches the same hierarchy as the "plates" key in master.yml.

jobber.yml comes after master.yml and its assignment loads after, so it overwrote the hierarchy plugins in master.yml. All of the relative plugins that depend on "job/shot/plates" now have a completely different mapping.

Now consider this

If one project has their WAYS_DESCRIPTORS set to this:

```
export WAYS_DESCRIPTORS=/path/to/master.yml
```

And another project includes the job-assignment folder:

```
export WAYS_DESCRIPTORS=/path/to/master.yml:/path/to/job/plugins/jobber.yml
```

The two projects could have completely different runtime behaviors despite having the exact same Python code.

Or maybe instead of having projects point to different files on disk, you have a job-based environment like this.

```
export WAYS_DESCRIPTORS=/jobs/$JOB/config/ways
```

Maybe one job is called "foo" and another is called "bar".

/jobs/foo/config/ways and /jobs/bar/config/ways could have different Plugin Sheet files customized for each job's needs.

With just a single, 8 line file, Ways's plugin structure can completely change.

2.4 Parsing Plugins

TODO

If you're reading this, it's because this page is still under construction. This section will eventually be filled with examples from production

Check back later!

2.5 API Details

The first thing to know about Ways is that it is not a tool, it is a toolkit. Ways is not immediately useful. You, the user, make Ways useful for your pipeline.

If you haven't already, please read the *API Summary* page on each of the main ideas that Ways uses. This document will expand on ideas written there.

2.5.1 Contexts

If you go to `ways.base.situation.Context`, you can read more about the class.

Context objects have

1. a hierarchy
2. a data property which is similar to a ChainMap
3. an actions property which can be used to extend a Context's interface

The Context object is the most important class in Ways because it encompasses two critical functions.

1. The Context is basically a "hierarchy interface". Everything that Ways builds is based on hierarchies that the user has full control over. Hierarchies each have their own actions, data, and order of importance and the Context lets us write functions around them.
2. The data that a Context uses is loaded exactly when it is queried. This means that we can load some plugins, create a Context object, add another plugin, and the Context picks up that new data without you doing anything.

Once you've read through the Context object, read `ways.api.get_context()`. You should become familiar with the Flyweight design pattern (http://sourcemaking.com/design_patterns/flyweight) because that's also an important part about how Ways creates Context objects.

2.5.2 Plugins

In the `ways.base.plugin` file, you'll find a couple very basic classes.

Context objects are built out of plugins. There's not much to say about plugins other than that they wrap around a dict that the user loads.

Two documents that cover all the different Plugin Sheet keyes are *Plugin Basics* and *Advanced Plugin Topics*.

At this point, it's a good idea to re-read each of Context object's methods and how those relate to the different plugin keys.

2.5.3 Ways Cache

Now that you understand Context objects, it's important to know how they are loaded. In the `ways.base.cache` file, you'll find the functions that are used to register Contexts and plugins, which we've talked about already, and also register Descriptors and Actions, which we haven't been explained yet.

When Ways first is imported and used, it gathers plugins that it can see in the WAYS_PLUGINS and WAYS_DESCRIPTOR environment variables. Once you're actually in Python, it's best to just use Python functions to add additional objects.

If you absolutely need to add paths to those environment variables, first ask yourself why you think you need to. If you still think its necessary, add the paths with os.environ and the run `ways.api.init_plugins()`.

Warning: This function **will** remove any objects that were added using Python so it's not recommended to use. But you can do it.

2.5.4 Descriptors

Read through *Descriptors* to get the jist about how Descriptor objects are built as a user. There's not much to say about them other than they're classes used to load Plugin Sheets. That way, you can load plugins into Ways from disk, a database, or whatever other method you'd like.

Other than that, they are not special in any way. Everything related to Descriptors is found in the `ways.base.descriptor` file. To see how they're loaded, revisit `ways.base.cache`.

In particular, two things in cache.py are interesting to maintainers.

1. add_search_path is just an alias to add_descriptor. The user can add plugins just by giving a filepath or folder and the Descriptor object needed will be built for them. Most of the time, that's all anyone need while using Ways.
2. add_descriptor and add_plugin both try their best to catch errors before they happen so the user can review any Descriptor or plugins that didn't load. For more information on that, check out *Troubleshooting Ways*.

2.5.5 Actions

Many pages talk about Actions. It's mentioned in *API Summary*, *Why use Ways*, *Common Patterns And Best Practices* and even has its own section in *Troubleshooting Ways*. There's not much point in repeating what has already been said so lets talk just about how Ways actually exposes Actions to the user.

When an Action is registered to Ways (using `ways.base.cache.add_action()`), the user specifies a hierarchy for the Action and a name to call it.

This is kept in a dictionary in `ways.ACTION_CACHE`.

When the user calls an action using `ways.api.Context.actions`, the following happens:

1. Ways looks up to see if that Action/Context has a definition for that Action. If there's no definition, look for a default value. If neither, raise an AttributeError.
2. If an Action is found, the function is wrapped using funtools.partial. The partial function adds the Context/Asset that called it as the first arg.

```
context = ways.api.get_context('something')
context.actions.some_action_name()
```

So by using functools.partial, we eliminate the need for the user to write

```
context.actions.some_action_name(context)
```

Any class that inherits from *ways.api.Action* is automatically registered to Ways, because the `ways.parsing.resource.ActionRegistry` metaclass registers the class once it's defined.

2.5.6 Assets

The Asset object is a simple wrapper around a Context object. Nearly all of its methods are used for getting data that the user has provided.

All classes and functions are located in the *ways.parsing.resource* file.

There are a couple functions in particular that are interesting to developers. The first is `ways.parsing.resource._get_value()`. If a user queries a part of an Asset that exists, the value is returned. But if the value doesn't exist, Ways is still able to "build" the value based on surrounding information. For the sake of making it easier to search for, the two methods are called "Parent-Search" and "Child-Search". All of the functions related to those search methods are either scoped functions in `ways.parsing.resource._get_value()` or somewhere within *ways.parsing.resource*.

The other function that's very important is `ways.parsing.resource._find_context_using_info()`.

Basically, if a user tries to run *ways.api.get_asset()* without giving a context, this function will try to "find" a matching Context to use instead. At the risk of reiterating the same information twice, read through `ways.parsing.resource._find_context_using_info()` and func:*ways.api.get_asset* docstrings. Both functions go in detail about the common pitfalls of auto-finding Contexts.

2.5.7 api.py

This module is where almost every function or class meant to be used by developers is put. There's nothing really special about it, just know that it's there and exists for the user's convenience.

2.6 Troubleshooting Ways

2.6.1 Loading Descriptors And Plugin Sheets

Descriptors aren't guaranteed to always load for a number of reasons. Depending on the class of the Descriptor and your input, there could be multiple reasons. Luckily, Ways keeps track of Descriptor objects that it tries to load so you can review a live session to find out what kind of errors you have.

Descriptor Failed To Import

Ways is designed to allow users to write custom Descriptor objects if they want. that. But if the user gives an import string that isn't on the PYTHONPATH, there's nothing that Ways can do to fix it.

Take a simple Descriptor dict, for example. If we convert it to a URL-encoded string, it looks like this:

```python
from urllib import parse

info = {
    "items": "/some/folder/path",
    "create_using": "foo.bar.bad.import.path",
    "uuid": "foo_bad_path"
}
parse.urlencode(info, doseq=True)
```

```
export WAYS_DESCRIPTORS=items=%2Fsome%2Ffolder%2Fpath&create_using=foo.bar.bad.import.
↪path&uuid=foo_bad_path
```

While this URL string is valid with no syntax errors, it will fail because the value for "create_using" doesn't exist.

This error can also come up if the URL-encoded string isn't correctly formatted (example: an encoding syntax error will also raise an error).

If you think your Descriptor failed to load and would like to check, search for the Descriptor using its UUID.

```
result = ways.api.trace_all_load_results()['descriptors']['my_uuid_here']
# Result:
# {
#     "status": "failed",
#     "item": "items=%2Fsome%2Ffolder%2Fpath&create_using=foo.bar.bad.import.path&
↪uuid=foo_bad_path",
#     "reason": "resolution_failure"
#     'traceback': some_traceback_info_here,
# }
```

ways.api.trace_all_load_results() will only be useful if you defined UUIDs for your Descriptor. If you don't create a Descriptor with a UUID, Ways will just create one for you but it will be random each time. You'll still have to iterate over all of the loaded Descriptors to find the one you want.

```
for result in ways.api.trace_all_load_results()['descriptors']:
    # ... do something to find the Descriptor you wanted

for result in ways.api.trace_all_load_results()['descriptors'].values():
    # ... do something to find the Descriptor you wanted
```

Descriptor has no method to get plugins

If the Descriptor loads but the object that Ways creates doesn't have a method for getting plugins, there's a very high chance that the Descriptor is will break on-load. To be on the safe side, Ways doesn't add the Descriptor to the system since it isn't sure about it and errors out, instead.

```
# cat /some/module_here.py
class BadDescriptor(object):

    '''A Descriptor that does not work.'''

    def __init__(self, items):
        '''Just create the object and do nothing else.'''
        super(BadDescriptor, self).__init__()

        self.get_plugins = None
```

Assuming module_here.py is on the PYTHONPATH, Ways can import it but it won't work because get_plugins isn't a callable function.

```
{
    "create_using": "module_here.BadDescriptor",
    "uuid": "some_uuid",
    "items": "/something/here"
}
```

And finally, that becomes

```
items=%2Fsomething%2Fhere&create_using=module_here.BadDescriptor&uuid=some_uuid
```

In this example, BadDescriptor is not callable and does not have a "get_plugins" method. Ways has no way of knowing how to get the plugins out of the Descriptor.

See *Descriptor Objects* for details on how to best build Descriptor objects.

2.6.2 Loading Standalone Plugins

Standalone plugins are Python files that load separately from the standard "Descriptor/Plugin Sheet" process. They're completely open - users can write whatever they want. But because of that, standalone plugins have more opportunities to fail.

Plugin Fails to Import

Finding out if Plugin files fail to import has almost the same syntax as a Descriptor.

```
export WAYS_PLUGINS=/some/path/that/doesnt/exist.py
```

Import failures are notoriously annoying because, even if the plugin has a uuid defined, Ways can't gather it if the module cannot import. Just like Descriptors, you'll have to iterate over each plugin result to find the ones that you're looking for.

```
failed_plugins = [item for item in ways.api.trace_all_plugin_results() if
                  item.get('reason') == ways.api.IMPORT_FAILURE_KEY]
```

Plugin "main()" Function is broken

If the Plugin has a "main()" function and running it causes some kind of error, that is also logged. Though this time, we can grab the Plugin by its uuid as long as it's defined in the file.

```
# cat /some/plugin.py
import ways.api

WAYS_UUID = 'some_uuid_here'

def main():
    raise ValueError('invalid main function')
```

In another file or a live Python session, we can search for this Plugin file's result.

```
result = ways.api.trace_all_plugin_results_info()['some_uuid_here']
```

2.6.3 Working In A Live Session

Depending on how complex your setup becomes or the number of people on your team, it may get difficult to keep track of the Contexts and Actions that are available to you while you begin to start working.

In most scenarios, you'll want to know what hierarchies you can use, what Contexts are available, and the Actions that those Context objects can use.

Working With Hierarchies

The first thing you'll want to know while working is what hierarchies that you can use.

Note: For the sake of completeness, the rest of the examples on this page will all refer to the plugins defined in this Plugin Sheet.

```
cat some_plugin_sheet.yml

plugins:
    a_plugin_root:
        hierarchy: foo
        mapping: /jobs
    another_plugin:
        hierarchy: foo/bar
        mapping: /jobs/foo/thing
    yet_another_plugin:
        hierarchy: foo/bar/buzz
    still_more_plugins:
        hierarchy: foo/fizz
    did_you_know_camels_have_three_eyelids?:
        hierarchy: foo/fizz/something
    okay_maybe_you_knew_that:
        hierarchy: foo/fizz/another
    but_I_thought_it_was_cool:
        hierarchy: foo/fizz/another/here
```

To get all hierarchies

```
ways.api.get_all_hierarchies()
# Result: {('foo', ), ('foo', 'bar'), ('foo', 'bar', 'buzz'),
#           ('foo', 'fizz'), ('foo', 'fizz', 'something'),
#           ('foo', 'fizz', 'another'), ('foo', 'fizz', 'another', 'here')}
```

To get hierarchies as a dictionary tree

```
ways.api.get_all_hierarchy_trees(full=True)
# Result:
# {
#     ('foo', ):
#     {
#         ('foo', 'bar'):
#         {
#             ('foo', 'bar', 'buzz'): {},
#         },
#         ('foo', 'fizz'):
#         {
```

```
#              ('foo', 'fizz', 'something'): {},
#              ('foo', 'fizz', 'another'):
#              {
#                  ('foo', 'fizz', 'another', 'here'): {}
#              },
#          },
#      },
# }
```

Or if you'd prefer a more concise version

```
ways.api.get_all_hierarchy_trees(full=False)
# Result:
# {
#     'foo':
#     {
#         'bar':
#         {
#             'buzz': {},
#         },
#         'fizz':
#         {
#             'something': {},
#             'another':
#             {
#                 'here': {}
#             },
#         },
#     },
# }
```

Once you've got a Ways object such as an Asset, Context, or just a simple hierarchy, you can also query "child" hierarchies from that point. A child hierarchy is any hierarchy that contains the given hierarchy.

```
hierarchy = ('foo', 'fizz')
context = ways.api.get_context(hierarchy)
asset = ways.api.get_asset({}, context=context)

# All three functions create the same output
ways.api.get_child_hierarchies(hierarchy)
ways.api.get_child_hierarchies(context)
ways.api.get_child_hierarchies(asset)
# Result: {('foo', 'fizz', 'something'), ('foo', 'fizz', 'another'),
                ('foo', 'fizz', 'another', 'here')}
```

And you can visualize it as a tree, too.

```
ways.api.get_child_hierarchy_tree(('foo', 'fizz'), full=True)
# Result:
#    {
#        ('foo', 'fizz', 'something'): {},
#        ('foo', 'fizz', 'another'):
#        {
#            ('foo', 'fizz', 'another', 'here'): {},
#        },
#    }
```

Note: The hierarchies that these functions return can be used to create Context objects assuming that there's at least one valid plugin in each hierarchy.

Working With Contexts

Context objects have different ways for resolving its Plugin objects. For example, *ways.api.Context.get_mapping_details()* resolves completely differently than *ways.api.Context.get_platforms()* or *ways.api.Context.get_mapping()* or even *ways.api.Context.get_max_folder()*.

When you get back a value that you didn't expect, it's always one of two problems. Either the Context didn't load the plugins that you expected or the plugins that were loaded didn't resolve the way you expected.

Checking The Loaded Context Plugins

Getting every Plugin that is loaded into Ways is a single command.

```
ways.api.get_all_plugins()
```

If you don't see the plugin that you're looking for in that list, it's possible that it was not found by the Descriptor that you thought it was. Once it's clear that all the Plugin objects needed are loaded into Ways, the last step is just to make sure that your Context is loading your Plugins.

Not all Plugin objects are loaded by a Context. For example, if a Plugin's *ways.api.DataPlugin.get_platforms()* method doesn't return the current user's platform, it is excluded. This Plugin-filtering lets Ways have Plugins with the same hierarchy but conflicting mappings coexist. It also lets the user define relative plugins so that Plugins meant for MacOS aren't loaded on Windows.

To get the raw list of Plugins that a Context can choose from, there is the *ways.api.Context.get_all_plugins()* method

```
context = ways.api.get_context('foo/bar')
raw_plugins = context.get_all_plugins()
plugins = context.plugins
unused_plugins = [plugin for plugin in raw_plugins if plugin not in plugins]
```

ways.api.get_all_plugins() shows you every Plugin that a Context can use. The "plugins" property shows you which of those Plugins were actually used and you can get the unused Plugin list by taking the difference between the two.

Checking Method Resolution

This section assumes that you've read *Plugin Basics*. It's important to know how Context objects resolve their plugins before starting to troubleshoot values that you may not expect.

```
context = ways.api.get_context('foo/bar')
ways.api.trace_method_resolution(context.get_mapping)
# Result: ['/jobs', '/jobs/foo/thing']

# To include the Plugins that created some output, use plugins=True
ways.api.trace_method_resolution(context, 'get_platforms' plugins=True)
# Result: [('/jobs', DataPlugin('etc' 'etc')),
#          ('/jobs/foo/thing', DataPlugin('etc', 'etc', 'etc'))]
```

ways.api.trace_method_resolution() works by taking the Context from its first plugin, running the given method, then uses the first 2 plugins and runs the given method again until every plugin that the Context sees has been run.

That way, it's obvious which plugin was loaded at what point and that plugin's effect on the method.

Working With Actions

Depending on what information you're working with, Actions can be queried in a few ways.

If you have a Context and you want to know what Actions that it is allowed to use, all you have to do is "dir" the "actions" property.

```
context = ways.api.get_context('foo/bar')
dir(context.actions)
# Result: ['action_names', 'here', 'and', 'functions', 'you', 'can', 'use']

# Assets work the same way
asset = ways.api.get_asset({'INFO': 'HERE'}, 'foo/bar')
dir(asset.actions)
# Result: ['action_names', 'here', 'and', 'functions', 'you', 'can', 'use']
```

Sometimes all you have is the name of an Action and aren't sure what hierarchies can use it.

```
# Get all of the hierarchies that allowed to use "some_action_name"
hierarchies = ways.api.get_action_hierarchies('some_action_name')

# To get the hierarchies for every action, use get_all_action_hierarchies
everything = ways.api.get_all_action_hierarchies()
```

Note: *ways.api.get_action_hierarchies()* will return every Action that matches the given Action name. So if multiple classes/functions are all registered under the same name, then every hierarchy that those Actions use will be returned. However, if a object like a function or class that was registered, only that object's hierarchies will be returned.

CHAPTER 3

Appendices

3.1 Common Patterns And Best Practices

While designing and working with Ways, a few re-occuring ideas would appear in production code over and over. This page is a collection of some of those good ideas.

3.1.1 Best Practices

This section is a series of things to include while writing Ways objects that are generally good ideas to do.

Writing mapping and mapping_details

Include a mapping for your plugins whenever possible. If you have some kind of information, a string or a dict, and you don't know what Context hierarchy it belongs to, mapping and mapping_details are used to "auto-find" the right Context.

Auto-find using mapping

Whenever you have to autofind a Context using *ways.api.get_asset()*, it's best to give a string whenever you can because then Ways can exact-match the string to a mapping, like this:

```
plugins:
    something:
        hierarchy: foo
        mapping: /jobs/{JOB}/shots
```

```
value = '/jobs/someJobName_12391231/shots'
asset = ways.api.get_asset(value)
```

If the mapping of the hierarchy you're looking for has at least one Token, you can give a dict:

```
value = {'JOB': 'someJobName_12391231'}
asset = ways.api.get_asset(value)
```

There's a pretty obvious problem with that though. If two hierarchies have a mapping that both use the "JOB" Token, Ways will try to return them both, which will cause an error.

```
plugins:
    something:
        hierarchy: foo
        mapping: /jobs/{JOB}/shots
    another:
        hierarchy: bar
        mapping: generic.{JOB}.string.here
```

```
value = {'JOB': 'someJobName_12391231'}
asset = ways.api.get_asset(value)  # Will raise an exception
```

Both "foo" and "bar" hierarchies use the JOB Token so Ways doesn't know which one to use.

The good news is, there is a way to distinguish between "foo" and "bar" in this worst-case scenario. Just describe "JOB" using mapping_details.

```
plugins:
    something:
        hierarchy: foo
        mapping: /jobs/{JOB}/shots
        mapping_details:
            JOB:
                parse:
                    regex: '\d+'
    another:
        hierarchy: bar
        mapping: generic.{JOB}.string.here
        mapping_details:
            JOB:
                parse:
                    regex: '\w+'
```

```
value = {'JOB': 'someJobName_12391231'}
asset = ways.api.get_asset(value)
asset.get_hierarchy()
# Result: 'foo'
```

Because the "foo" hierarchy was defined with regex and it expected some integer, and "bar" is allowed to have non-digit characters, Ways was able to figure out which Context to use for our Asset.

In short, it's a good idea to define mapping and mapping_details basically always.

Add a UUID

In Ways, the UUID is an optional string that you can add to every plugin. This UUID is useful for searching and debugging so it's a good idea to include it whenever you can.

```
plugins:
    hierarchy: foo
    uuid: some_string_that_is_not_used_anywhere_else
```

A UUID must be unique, even in other Ways-related files. If the same UUID comes up more than once, Ways will raise an exception to let you know.

Filepaths and mapping

If you use Ways for filepaths, make sure to enable the "path" key to avoid OS-related issues.

```
plugins:
    path_out:
        hierarchy: foo
        mapping: /etc/some/filepath
        path: true
```

The reason to do this has explained in *path* so head there if further explanation is needed.

Action Patterns

By now you should know about Actions (If not, read through this *Extend Ways Using Actions*). Actions are how Ways extends its objects with additional functions.

Because Actions are applied to certain hierarchies, sometimes you may call an Action on an Asset or Context that you think exists but doesn't. When that happens, AttributeError is raised.

```
plugins:
    foo:
        hierarchy: some/hierarchy
    another:
        hierarchy: action/hierarchy
```

```
class ActionOne(ways.api.Action):

    name = 'some_action'

    @classmethod
    def get_hierarchy(cls):
        return 'some/hierarchy'

    def __call__(self, obj):
        return ['t', 'a', 'b', 'z']


class ActionTwo(ways.api.Action):

    name = 'some_action'

    @classmethod
    def get_hierarchy(cls):
        return 'action/hierarchy'

    def __call__(self, obj):
        return [1, 2, 4, 5.4, 6, -2]

for hierarchy in ['some/hierarchy', 'action/hierarchy', 'bar']:
    context = ways.api.get_context(hierarchy)
    context.actions.some_action()
```

This will cause you to want to write lots of code using try/except:

```
try:
    value = context.actions.some_action()
except AttributeError:
    value = []
```

A better way is to assign a default value for your Action. This value will get returned whenever you call a missing Action.

In a plugin file, you can write this:

/some/plugin/defaults.py

```
import ways.api

class ActionTwo(ways.api.Action):

    name = 'some_action'

    @classmethod
    def get_hierarchy(cls):
        return 'action/hierarchy'

    def __call__(self, obj):
        return [1, 2, 4, 5.4, 6, -2]

def main():
    '''Add defaults for actions.'''
    ways.api.add_action_default('some_action', [])
```

Then add the path to /some/plugin/defaults.py to your WAYS_PLUGINS environment variable.

Now, in any file you'd like, you can work as normal.

```
import ways.api

context = ways.api.get_context('foo/bar')
for item in context.actions.some_action():
    # ...
```

To summarize, it's usually a good idea to define a default value in the same file that defines Actions. That way there is always a fallback value.

Note: If you want certain hierarchies to have different default values, specify a hierarchy while you define your default value.

ways.api.add_action_default('some_action', [], hierarchy='foo/bar')

3.1.2 Designing Plugins

Appending vs Defining

It's mentioned in several other pages such as *path* and *Appending To Plugins* but you have 3 options to add information to hierarchies. You can either just add the information to the original plugin or append to it, using another absolute plugin or a relative plugin.

```
plugins:
    root:
        hierarchy: foo
    another:
        hierarchy: bar
        mapping: a_mapping
    absolute_append:
        hierarchy: foo
        data:
            something_to_add: here
    relative_append:
        hierarchy: ''
        mapping: something
        path: true
        uses:
            - foo
            - bar
```

In this example, the “absolute_append” plugin will append to “root” and “relative_append” appends to “root” and “another” at once. If you need better control over your plugins, using absolute appends will tend to be a very clear, simple way to do it. If you need to make a broad change to many plugins at once, relative appends make more sense to do since you can specify many plugins and add information all in one plugin.

Relative appends have one problem though - you can’t customize what gets appended to both hierarchies.

In the above example, mapping and path are both appending onto “root” and “another”. But say for example you only wanted mapping to append to “root” and not to “another”? It’s not possible - you’d have to split the relative plugin into two relative plugins. At that point, you might as well use absolute appends.

It’s a balancing act and you’ll find yourself gravitating to one style or another.

3.1.3 Asset Swapping

Ways comes with an object called Asset (*ways.api.Asset*) that is used for basic asset management. If you have your own classes that you’d prefer to use instead, adding those objects to Ways is fairly simple.

Register A Custom Class

An generic Ways Asset expects at least two arguments, the object that represents the information to pass to the Asset and the Context that does with that that information. The Context is optional, as mentioned before.

```
info = {'foo': 'bar'}
context = 'some/thing/context'
ways.api.get_asset(info, context)
```

If you have a class that takes two or more arguments, you can use that class directly in place of an Asset.

```
import ways.api

class SomeNewAssetClass(object):

    '''Some class that will take the place of our Asset.'''

    def __init__(self, info, context):
        '''Create the object.'''
        super(SomeNewAssetClass, self).__init__()
        self.context = context

    def example_method(self):
        '''Run some method.'''
        return 8

    def another_method(self):
        '''Run another method.'''
        return 'bar'
context = ways.api.get_context('some/thing/context')
ways.api.register_asset_class(SomeNewAssetClass, context)
asset = ways.api.get_asset({'JOB': 'something'}, context='some/thing/context')
asset.example_method()
# Result: 8
```

If the class isn't designed to work with Ways or takes 0 or 1 arguments, you can still use it. Just add an init function:

```
import ways.api

class SomeNewAssetClass(object):

    '''Some class that will take the place of our Asset.'''

    def __init__(self):
        '''Create the object.'''
        super(SomeNewAssetClass, self).__init__()

def custom_init(*args, **kwargs):
    return SomeNewAssetClass()

def main():
    '''Register a default Asset class for 'some/thing/context.'''
    context = ways.api.get_context('some/thing/context')
    ways.api.register_asset_class(
        SomeNewAssetClass, context=context, init=custom_init)
```

By default, you will need to register a class/init function for every hierarchy that you want to swap. So if you had hierarchies like this, “some”, “some/other”, “some/other/child”, and “some/other/child/hierarchy” then you'd need to register the custom class for all 4 hierarchies individually. If you're prefer to register them for “this hierarchy and all its subhierarchies”, set children to True.

```
ways.api.register_asset_class(SomeNewAssetClass, context='some', children=True)
```

CHAPTER 4

Modules

4.1 API Summary

The API Summary is a brief description of the major parts of Ways.

Ways is an API that helps developers define and use Context and Asset objects. Contexts and Assets aren't directly created. Instead, they're generated by Ways, automatically, from plugins defined in a Plugin Sheet.

Your job as the user is to define plugins. Those plugins will describe Contexts. If needed, Contexts and Plugins can be defined and registered manually but this isn't the default behavior of Ways.

4.1.1 Plugin Sheets

Plugins Sheets are files on disk or on a server that describe a Context. Once a single plugin in a single Plugin Sheet is created, it can be immediately used. Adding more plugins will append to the existing Context or create more Contexts.

Plugin Sheets can be JSON, YAML, or Python files.

4.1.2 Descriptor Objects

A Descriptor is a class or function used to load Plugin Sheets. Whenever a file or folder path is added to WAYS_DESCRIPTORS, Ways generates Descriptor objects in the background.

For most people, just knowing that Descriptors exist is all they'll need to know.

That said, the Descriptor objects that ship with Ways only cover filepaths and local/remote git repositories. If you need something special, check out *Descriptors* to learn how Descriptors work to create your own.

4.1.3 Plugins

Descriptor objects determine how Plugin Sheets load. The ones that Ways ship with recognize JSON, YAML, and Python files in alphabetical order but it can be changed to whatever you want by creating and using your own Descriptor objects.

Whenever a user tries to create a Context, the Context's Plugin objects are looked up as read-only data, combined, and then hooked into the Context.

4.1.4 Context Objects

Context objects are containers of metadata and Plugin information.

Plugins are loaded into a Context on-demand - A Plugin Sheet can be defined, and afterwards a Context object could be instantiated, and then more Plugin Sheets / Descriptors could be added in a single interactive session and the original Context you created will already have all the new Plugin settings.

(It's not recommended to do this but Ways will allow it).

There's a ton of things that you can use Context objects to do but to keep this page short, the examples will stop here. Go to *Getting Started* to try it for yourself if you're ready.

4.1.5 Asset Objects

Once you've worked with them for a short while, it'll become obvious that Context objects are pretty limited. Also, every hierarchy is only ever allowed one instance of a Context which further limits how useful they can be.

An Asset object is basically the same as a Context object but you can instantiate new version of them any time you'd like. Also, Asset objects contain extra information about the Context. The Asset object is a very minimal implementation of an Asset Management System (AMS) and is heavily relies on Actions and its Context to do the heavy-lifting.

4.1.6 Action Objects

Actions are classes or functions that attach to a Context or Asset. Actions can be defined ahead of the Context/Asset objects that they act upon and do completely different things for different hierarchies.

Head over to *getting_started* to learn about creating Descriptors, Contexts, Actions, and more.

4.2 ways package

Ways is split into two main sections. `ways` and `ways.api`. Of the two, 99% of all your work is going to use classes and functions out of `ways.api` but in the exceptional case where you need to do something special, You'd use the parent module's functions.

4.2.1 Main Module

ways.api is where almost every function and class is added to and contains everything that you'd need to work with Ways.

ways.api module

Expose common functionality.

This module's responsibility to maintain backwards and forwards compatibility so that this package can be refactored without breaking any tools.

It's recommended to always import and use modules, here.

`ways.api.`**`decode`**(*obj*)
dict[str]: Convert a URL-encoded string back into a dict.

`ways.api.`**`encode`**(*obj*)
Make the given descriptor information into a standard URL encoding.

Parameters

- **`obj`** (*`dict[str]`*) – The Descriptor information to serialize.
- **`is normally something like`** (*`This`*) –
- **`{'create_using'`** – ways.api.FolderDescriptor}.

Returns The output encoding.

Return type str

`class` `ways.api.`**`Plugin`**
Bases: `object`

An add-on that is later retrieved by Context to gather its data.

`add_to_registry = True`

`data`
**dict[str]* – The display properties (like {'color'* – 'red'}).

`class` `ways.api.`**`DataPlugin`**(*name*, *sources*, *info*, *assignment*)
Bases: *`ways.base.plugin.Plugin`*

An add-on that was made from a serialized file (JSON/YAML/etc).

This class behaves exactly like a regular Plugin object and is stored in the same space as Plugin objects.

DataPlugin does not add itself to the cache automatically. It is the responsibility of some other class/function to register it to Ways.

We do this so that we can have better control over the DataPlugin's args and its assignment before it hits the cache.

`add_to_registry = False`

`get_assignment`()
str: Where this Plugin lives in Ways, along with its hierarchy.

`get_groups`()
Get the groups that this Plugin evaluates onto.

Note: The term 'groups' is not the same as the assignment of a Plugin. They are two different things.

Returns The groups.

Return type tuple[str]

`get_hierarchy`()
tuple[str] or str: The location that this Plugin exists within.

`get_mapping`()
str: The physical location of this Plugin (on the filesystem).

get_mapping_details()
dict[str]: Information about the mapping, if needed.

get_max_folder()
str: The furthest location up that this plugin can navigate to.

get_platforms()
set[str]: The platforms that this Plugin is allowed to run on.

get_uses()
tuple[str]: The Context hierarchies this instance depends on.

get_uuid()
str: A unique ID for this plugin.

is_path()
If the mapping is a filepath or None if unsure.

Returns If the mapping is a path to a file/folder on disk.

Return type bool or NoneType

class ways.api.**Context**(*hierarchy*, *assignment=''*, *connection=None*)
Bases: object

A collection of plugins that are read in order to resolve its methods.

as_dict(*changes=True*)
Convert this object into a dictionary.

This is different from a standard repr(Context) because it will include items that are not part of the Context's initialization.

It also creates a deepcopy of its contents, so that any changes to this dictionary won't affect the original object.

Parameters **changes** (bool, optional) – If True, the output will contain original plugin data as well as any changes that the user made over top of the original. If False, only the original information is returned. Default is True.

Returns A copy of the current information of this class.

Return type dict[str]

checkout(*assignment='master'*)
Make a new Context instance and return it, with the same hierarchy.

Parameters **assignment** (str, optional) – The new assignment to get.

Returns The new Context object.

Return type *ways.api.Context*

data
dict[str] – Data that was automatically generated and user data.

get_action(*name*)
ways.api.Action or callable or NoneType: The Action.

get_all_plugins(*hierarchy=''*, *assignment=''*)
list[*ways.api.Plugin*]: The found plugins, if any.

get_all_tokens()
Get the tokens in this Context's mapping and any subtokens.

Subtokens are tokens that are inside another token's mapping.

Returns All of the tokens known to this Context.

Return type set[str]

get_assignment()
str: The assignment for this Context.

get_groups()
tuple[str]: The groups that this Context belongs to.

get_hierarchy()
tuple[str]: The path to this Context.

get_mapping()
str: The mapping that describes this Context.

get_mapping_details()
Get the information that describes a Context instance's mapping.

This function is the same as "mapping_details" key that you'd see in a Plugin Sheet file and is critical to how a Context's parser builds into a file path.

Without it, you cannot get a proper filepath out of a Context.

Returns The information.

Return type dict[str]

get_mapping_tokens(*mapping=''*)
list[str]: Get all of the tokens that are in this Context's mapping.

get_max_folder()
str: The highest mapping point that this Context lives in.

get_parser()
`ways.api.ContextParser`: A parser copy that points to this Context.

get_platforms()
Get The OSes that this Context runs on.

The recognized platforms for this method is anything that platform.system() would return. (Examples: ['darwin', 'java', 'linux', 'windows']).

Returns The platforms that this Context is allowed to run on.

Return type set[str]

get_str(**args*, ***kwargs*)
Get the Context's mapping as filled-out text.

Parameters

- ***args** (*list*) – Positional args to send to ways.api.ContextParser.get_str.
- ****kwargs** (*dict*) – Keyword args to send to ways.api.ContextParser.get_str.

is_path()
bool: If the user indicated that the given mapping is a filepath.

plugins
Find all of the "valid" plugins for this instance.

What decides if a Plugin is "found" depends on a number of factors. First, the plugin needs to be inside the hierarchy of the Context, have the same assignment, and the platform(s) assigned to the Plugin must match our current system's platform. If a platform override is specified (aka if WAYS_PLATFORM has been set) then the Plugin object's platform has to match that, instead.

Raises

- `ValueError` – If the platform in WAYS_PLATFORM was invalid.
- `ValueError` – If the plugin found has a platform that is not found in our recognized_platforms variable.

Returns The found plugins.

Return type list[*ways.api.Plugin*]

revert()

Set the data on this instance back to its default.

classmethod validate_plugin(*plugin*)

Check if a plugin is "valid" for this Context.

Typically, a plugin is invalid if it was meant for a different OS (example), a Windows plugin shouldn't be added to a Context that is being run on a Linux machine.

Parameters **plugin** (*ways.api.Plugin*) – The plugin to check.

Raises

- `OSError` – If the user specified an unrecognized environment using the PLATFORM_ENV_VAR environment variable.
- `EnvironmentError` – If the plugin's environment does not match this environment.

Returns The plugin (completely unmodified).

Return type *ways.api.Plugin*

ways.api.**get_context**(*hierarchy*, *assignment=''*, *follow_alias=False*, *force=False*, **args*, ***kwargs*)

Get a persistent Context at some hierarchy/assignment location.

This function uses a Flyweight factory to manage the instance objects that it returns.

Reference: http://sourcemaking.com/design_patterns/flyweight http://sourcemaking.com/design_patterns/flyweight/python/1

Parameters

- **hierarchy** (*tuple[str]*) – The location to look for our instance.
- **assignment** (*str*) – The category/grouping of the instance. If no assignment is given, Ways will gather plugins in the order defined in the WAYS_PRIORITY environment variable and create plugins based on that.
- ***args** (*list*) – If an object instance is found at the hierarchy/assignment, this gets passed to the instantiation of that object.
- ****kwargs** (*dict[str]*) – If an object instance is found at the hierarchy/assignment, this gets passed to the instantiation of that object.

Returns An instance of that class. If the Context that is queried doesn't have any Plugin objects defined for it, it's considered 'empty'. To avoid faults in our code, we return None.

Return type <class_tuple instance> or NoneType

ways.api.**register_context_alias**(*alias_hierarchy*, *old_hierarchy*)

Set a hierarchy to track the changes of another hierarchy.

This function lets you refer to plugins and Context objects without specifying their full names.

Example

```
>>> class SomePlugin(plugin.Plugin):
>>>     def get_mapping(self):
>>>         return '/some/path/here'
```

```
>>>     def get_hierarchy(self):
>>>         return ('nuke', 'scenes')
```

```
>>>     def get_platforms(self):
>>>         return '*'
```

```
>>> class AnotherPlugin(plugin.Plugin):
>>>     def get_mapping(self):
>>>         return '/some/path/here'
```

```
>>>     def get_hierarchy(self):
>>>         return ('maya', 'scenes')
```

```
>>>     def get_platforms(self):
>>>         return '*'
```

```
>>> sit.register_context_alias('maya_scenes', 'maya/scenes')
>>> context = ways.api.get_context('maya_scenes')
>>> # The resulting Context object has the hierarchy ('maya_scenes', )
>>> # but has all of the plugins from 'maya/scenes'
```

```
>>> sit.register_context_alias('maya_scenes', 'nuke/scenes')
```

```
>>> # Now, the Context('maya_scenes') is pointing to a Nuke Context
>>> # we can immediately work with this Context without having to
>>> # re-instantiate the Context
```

Raises `ValueError` – If the alias is the same as the hierarchy that it's trying to be aliased to or if the alias was already defined.

`ways.api.`**`resolve_alias`**(*hierarchy*)

Get the real hierarchy that the given alias represents.

Parameters **`hierarchy`** (*`iter[str] or str`*) – The alias hierarchy to convert.

Returns The real hierarchy.

Return type tuple[str]

`ways.api.`**`clear_aliases`**()

Remove all the stored aliases in this instance.

`ways.api.`**`clear_contexts`**()

Remove every Context instance that this object knows about.

If a Context is re-queried after this method is run, a new instance for the Context will be created and returned.

Running this method is not recommended because it messes with the internals of Ways.

class `ways.api.`**`Asset`**(*info*, *context*, *parse_type='regex'*)

Bases: `object`

An object that contains a Context and data about the Context.

The idea of this class is to keep Context information abstract, and let Context parse/use that information. Depending on what the Context is for, it could be used to ground the information to a filesystem or a database or some other structure that the Context knows about.

`get_missing_required_tokens`()

Find any token that still needs to be filled for our parser.

If a token is missing but it has child tokens and all of the child tokens are defined, it is excluded from the final output. If the missing token is a child of some parent token that is defined, then the value of the token is parsed. If the parse is successful, the token is excluded from the final output.

Returns Any tokens that have no value.

Return type list[str]

`get_str`(*required=True*, **args*, ***kwargs*)

Get the full path to the asset, if any.

Parameters

- **`required`** (`bool`, optional) – If True and there are tokens that are required that still are not filled, raise an error. If False, return the incomplete string. Default is True.
- **`*args`** (*`list`*) – Positional args to send to ContextParser.get_str.
- **`**kwargs`** (*`list`*) – Keywords args to send to ContextParser.get_str.

Raises `ValueError` – If required is True (in other words, we assume that)

Returns The resolved string for this instance.

Return type str

`get_token_parse`(*name*, *parse_type=''*)

Get the parse expression for some token name.

Parameters

- **`name`** (*`str`*) – The name of the token to get parse details from.
- **`parse_type`** (`str`, optional) – The engine type whose expression will be returned. If no parse_type is given, the stored parse_type is used.

Returns The parse expression used for the given token.

`get_unfilled_tokens`(*required_only=False*)

Get the tokens in this instance that still don't have values.

Parameters **`required_only`** (`bool`, optional) – If True, do not return optional tokens. If False, return all tokens, required and optional. Default is False.

Returns The tokens that still need values.

Return type list[str]

`get_value`(*name*, *real=False*)

Get some information about this asset, using a token-name.

If the information is directly available, we return it. If it isn't though, it is searched for, using whatever information that we do have.

If the token name is a child of another token that is defined, we use the parent token to "build" a value for the token that was requested.

If the token name is a parent of some other tokens that all have values, we try to "build" it again, by combining all of the child tokens.

In both cases, the return value is created but not defined. But it lets you do this:

Example

```
>>> shot_info = {
...     'JOB': 'someJob',
...     'SCENE': 'SOMETHING',
...     'SHOT': 'sh0010'   # Pretend SHOT_NUMBER is a child of SHOT
... }
>>> shot_asset = resource.Asset(shot_info, context='job/scene/shot')
>>> shot_asset.get_value('SHOT_NUMBER')
... # Result: '0010'
```

Parameters

- **name** (*str*) – The token to get the value of.
- **real** (bool, optional) – If True, the original parsed value is returned. If False and the given token has functions defined in "before_return" then those functions will process the output and then return it. Default is False.

Returns The value at the given token.

set_value (*key*, *value*, *force=False*)
Store the given value to some key.

Parameters

- **key** (*str*) – The token that our value will be stored into.
- **value** (*str*) – The value to store.
- **force** (bool, optional) – If False, values are checked against their tokens before being set. If True, values are set for each token, even if they are not valid input for that token. Default is False.

ways.api.**get_asset** (*info*, *context=None*, **args*, ***kwargs*)
Get some class object that matches the given Context and wraps some info.

Parameters

- **info** (*dict[str] or str*) – The info to expand. If the input is a dict, it is passed through and returned. If it is a string, the string is parsed against the given context. Generally speaking, it's better to give a string that is an exact or partial match to a Context's mapping than it is to give a dict. This is doubly true if no context is given.
- **context** (*ways.api.Context* or str or tuple[str]', optional) – The Context to use for the asset. If a string is given, it is assumed to be the Context's hierarchy and a Context object is constructed. If nothing is given, the best possible Context is "found" and tried. This auto-find process will try to find the "best" match by looking at every known Context's mapping. A match is not guaranteed. Default is None.
- ***args** (*list*) – Optional position variables to pass to our found class's constructor.

- **`**kwargs`** (*`dict`*) – Optional keyword variables to pass to our found class's constructor.

Raises `NotImplementedError` – If context is None. There's no auto-find-context option yet.

Returns The found class object or NoneType. If no class definition was found for the given Context, return a generic Asset object.

`ways.api.`**`get_asset_class`**(*hierarchy*)
Get the class that is registered for a Context hierarchy.

`ways.api.`**`get_asset_info`**(*hierarchy*)
Get the class and initialization function for a Context hierarchy.

Parameters **`hierarchy`** (*`tuple[str] or str`*) – The hierarchy to get the asset information of.

Returns The class type and the function that is used to instantiate it.

Return type tuple[classobj, callable]

`ways.api.`**`register_asset_class`**(*class_type*, *context*, *init=None*, *children=False*)
Change get_asset to return a different class, instead of an Asset.

The Asset class is useful but it may be too basic for some people's purposes. If you have an existing class that you'd like to use with Ways,

Parameters

- **`class_type`** (*`classobj`*) – The new class to use, instead. context (str or *`ways.api.Context`*): The Context to apply our new class to.
- **`init`** (`callable`, optional) – A function that will be used to create an instance of class_type. This variable is useful if you need to customize your class_type's __init__ in a way that isn't normal (A common example: If you want to create a class_type that does not pass context into its __init__, you can use this variable to catch and handle that).
- **`children`** (`bool`, optional) – If True, this new class_type will be applied to child hierarchies as well as the given Context's hierarchy. If False, it will only be applied for this Context. Default is False.

`ways.api.`**`reset_asset_classes`**(*hierarchies=()*)
Clear out the class(es) that is registered under a given hierarchy.

Parameters **`hierarchies`** (*`iter[tuple[str]]`*) – All of the hierarchies to remove custom Asset classes for. If nothing is given, all hierarchies will be cleared.

`ways.api.`**`Action`**
alias of `_Aktion`

`ways.api.`**`add_action`**(*action*, *name=''*, *context=''*, *assignment='master'*)
Add a created action to this cache.

Parameters

- **`action`** (*`ways.api.Action`*) – The action to add. Action objects are objects that get passed a Context object and run a function.
- **`name`** (`str`, optional) – A name to use with this action. The name must be unique to this hierarchy/assignment or it risks overriding another Action that might already exist at the same location. If no name is given, the name on the action is tried, instead.
- **`context`** (*`ways.api.Context`* or str) – The Context or hierarchy of a Context to add this Action to.
- **`assignment`** (`str`, optional) – The group to add this action to, Default: 'master'.

Raises

- `RuntimeError` – If no hierarchy is given and no hierarchy could be found on the given action.
- `RuntimeError` – If no name is given and no name could be found on the given action.

`ways.api.`**`trace_all_load_results`**`()`

Get the load results of every plugin and descriptor.

If the UUID for a Descriptor cannot be found, Ways will automatically assign it a UUID.

Using this function we can check 1. What plugins that Ways found and tried to load. 2. If our plugin loaded and, if not, why.

Returns The main dictionary has two keys, "descriptors" and "plugins". Each key has an OrderedDict that contains the UUID of each Descriptor and plugin and their objects.

Return type dict[str, `collections.OrderedDict` [str, dict[str]]]

`ways.api.`**`trace_actions`**`(`*obj*, **args*, ***kwargs*`)`

Get actions that are assigned to the given object.

Parameters

- **obj** (*ways.api.Action* or *ways.api.AssetFinder* or *ways.api.Context* or *ways.api.Find*) – The object to get the actions of.
- ***args** (*list*) – Position args to pass to ways.get_actions.
- ****kwargs** (*dict[str]*) – Keyword args to pass to ways.get_actions.

Returns The actions in the hierarchy.

Return type list[*ways.api.Action* or callable]

`ways.api.`**`trace_action_names`**`(`*obj*, **args*, ***kwargs*`)`

Get the names of all actions available to a Ways object.

Parameters

- **obj** (*ways.api.Action* or *ways.api.AssetFinder* or *ways.api.Context* or *ways.api.Find*) – The object to get the action names of.
- ***args** (*list*) – Position args to pass to ways.get_action_names.
- ****kwargs** (*dict[str]*) – Keyword args to pass to ways.get_action_names.

Returns The names of all actions found for the Ways object.

Return type list[str]

`ways.api.`**`trace_method_resolution`**`(`*method*, *plugins=False*`)`

Show the progression of how a Context's method is resolved.

Parameters

- **method** (*callable*) – Some function on a Context object.
- **plugins** (`bool`, optional) – If False, the result at every step of the method will be returned. If True, the Plugin that created each result will be returned al along with the result at every step. Default is False.

Returns The plugin resolution at each step.

Return type list

`ways.api.`**`trace_actions_table`**(*obj*, **args*, ***kwargs*)
Find the names and objects of every action registered to Ways.

Parameters

- **`obj`** (*`ways.api.Action`* or *`ways.api.AssetFinder`* or *`ways.api.Context`* or *`ways.api.Find`*) – The object to get the available actions table of.
- **`*args`** (*`list`*) – Position args to pass to ways.get_actions_info..
- **`**kwargs`** (*`dict[str]`*) – Keyword args to pass to ways.get_action_info.

Returns The names and actions of an object.

Return type dict[str, *`ways.api.Action`* or callable]

`ways.api.`**`trace_assignment`**(*obj*)
str: Get the assignment for this object.

`ways.api.`**`trace_context`**(*obj*)
Get a Context, using some object.

This function assumes that the given object is a Ways class that only has 1 Context added to it (not several).

Parameters **`obj`** – Some Ways object instance.

Returns The found Context.

Return type *`ways.api.Context`* or NoneType

`ways.api.`**`trace_hierarchy`**(*obj*)
Try to find a hierarchy for the given object.

Parameters **`obj`** (*`ways.api.Action`* or *`ways.api.AssetFinder`* or *`ways.api.Context`* or *`ways.api.Find`*) – The object to get the hierarchy of.

Returns The hierarchy of some object.

Return type tuple[str]

`ways.api.`**`get_action_hierarchies`**(*action*)
Get the Context hierachies that this Action is registered for.

Note: get_action_hierarchies will return every Action that matches the given Action name. So if multiple classes/functions are all registered under the same name, then every hierarchy that those Actions use will be returned. However, if a object like a function or class that was registered, only that object's hierarchies will be returned.

Parameters **`action`** (*`str or class or callable`*) – The action to get the hierachies of.

Returns The hierarchies for the given Action.

Return type set[tuple[str]]

`ways.api.`**`get_all_action_hierarchies`**()
Organize every Action that is registered into Ways by object and hierarchy.

Returns

dict[str: str or set]]: Actions are stored as either classes or functions. Each Action's value is a dict which contains the hierachies that the Action is applied to and its registered name.

Return type dict[class or callable

ways.api.**get_all_hierarchies**()
set[tuple[str]]: The Contexts that have plugins in our environment.

ways.api.**get_child_hierarchies**(*hierarchy*)
list[tuple[str]]: Get hierarchies that depend on the given hierarchy.

ways.api.**get_child_hierarchy_tree**(*hierarchy*, *full=False*)
Get all of the hierarchies that inherit the given hierarchy.

Examples

```
>>> get_all_hierarchy_trees(full=True)
>>> {
>>>     ('foo', ): {
>>>         ('foo', 'bar'): {
>>>             ('foo' 'bar', 'fizz'): {},
>>>         },
>>>         ('foo', 'something', 'buzz'): {
>>>             ('foo', 'something', 'buzz', 'thing'): {},
>>>         },
>>>     },
>>> }
```

```
>>> get_all_hierarchy_trees(full=False)
>>> {
>>>     'foo': {
>>>         'bar': {
>>>             'fizz': {},
>>>         },
>>>         'something': {
>>>             'buzz': {
>>>                 'thing': {},
>>>             },
>>>         },
>>>     },
>>> }
```

Parameters

- **hierarchy** (*tuple[str]*) – The hierarchy to get the child hierarchy items of.
- **full** (bool, optional) – If True, each item in the dict will be its own hierarchy. If False, only a single part will be written. See examples for details. Default is False.

Returns The entire hierarchy.

Return type collections.defaultdict[str]

ways.api.**get_all_hierarchy_trees**(*full=False*)
Get a description of every Ways hierarchy.

Examples

```
>>> get_all_hierarchy_trees(full=True)
>>> {
>>>     ('foo', ): {
```

```
>>>         ('foo', 'bar'): {
>>>             ('foo' 'bar', 'fizz'): {},
>>>         },
>>>         ('foo', 'something', 'buzz'): {
>>>             ('foo', 'something', 'buzz', 'thing'): {},
>>>         },
>>>     },
>>> }
```

```
>>> get_all_hierarchy_trees(full=False)
>>> {
>>>     'foo': {
>>>         'bar': {
>>>             'fizz': {},
>>>         },
>>>         'something': {
>>>             'buzz': {
>>>                 'thing': {},
>>>             },
>>>         },
>>>     },
>>> }
```

Parameters **full** (bool, optional) – If True, each item in the dict will be its own hierarchy. If False, only a single part will be written. See examples for details. Default is False.

Returns The entire hierarchy.

Return type `collections.defaultdict[str]`

class ways.api.**FileDescriptor**(*items*)

Bases: `object`

A generic class that creates Plugin objects from a file, on-disk.

Note: Any FileDescriptor class that returns back Plugin objects is valid (Descriptors can query from a database, locally on file, etc, etc. It's all good) except there is one major requirement. FileDescriptor-like objects cannot append asynchronously. In other words, If a FileDescriptor manages threads that each find Plugin objects and append to a list of Plugin objects whenever each thread finishes, the Plugin objects might append out of order - which will create results that will be hard to debug.

It's recommended to not use threading at all unless this return process is managed (like with a queue or some other kind of idea).

classmethod filter_plugin_files(*items*)

Only get back the files that are likely to be plugin files.

get_plugin_info(*path*)

Given some file path, get its metadata info.

Parameters **path** (*str*) – The path to some directory containing plugin objects.

Returns The information about this plugin path.

Return type dict[str]

get_plugins(*items=None*)

Get the Plugin objects that this instance is able to find.

Note: This object will sort the items it is given before retrieving its Plugin objects so files/folders that are given different priorities depending on how their paths are named.

Parameters **`items`** (*`iterable[str] or str`*) – The paths that this FileDescriptor looks for to find Plugin objects. If not items are given, the instance's stored items are used, instead.

Returns The plugins.

Return type list[*`ways.api.Plugin`*]

`classmethod get_supported_extensions``()`
list[str]: The Plugin file extensions.

`class` `ways.api.`**`FolderDescriptor`**`(`*items*`)`
Bases: *`ways.base.descriptor.FileDescriptor`*

A generic abstraction that helps find Plugin objects for our code.

Note: Any FileDescriptor class that returns back Plugin objects is valid (Descriptors can query from a database, locally on file, etc, etc. It's all good) except there is one major requirement. FileDescriptor-like objects cannot append asynchronously. In other words, If a FileDescriptor manages threads that each find Plugin objects and append to a list of Plugin objects whenever each thread finishes, the Plugin objects might append out of order - which will create results that will be hard to debug.

It's recommended to not use threading at all unless this return process is managed (like with a queue or some other kind of idea).

`class` `ways.api.`**`GitLocalDescriptor`**`(`*path*, *items*, *branch='master'*`)`
Bases: *`ways.base.descriptor.FolderDescriptor`*

An object to describe a local git repository.

This class conforms its input to files that its base class can read and then calls it. Otherwise that, it's not special.

`class` `ways.api.`**`GitRemoteDescriptor`**`(`*url*, *items*, *path=''*, *branch='master'*`)`
Bases: *`ways.base.descriptor.GitLocalDescriptor`*

A Descriptor that clones an online Git repository.

`class` `ways.api.`**`ContextParser`**`(`*context*`)`
Bases: `object`

A class that's used to fill out missing values in a Context's mapping.

Some quick terms: If you see the word 'token', it means a piece of a string that needs to be filled out, such as 'some_{TOKEN}_here'.

A field is the token + its information. For example, if the token being filled out is optional or if input to a token is valid.

This class is meant to expand and resolve the tokens inside the mapping of a Context object.

`get_all_mapping_details``()`
Get the combined mapping details of this Context.

Note: The "true" combined mapping details of our Context is actually just Context.get_mapping_details(). This method can produce different results because it is yielding/updating

Context.get_mapping_details() with all of its plugin's data. Use with caution.

Returns The combined mapping_details of our Context and plugins.

Return type dict[str]

get_child_tokens (*token*)

Find the child tokens of a given token.

Parameters **token** (`str`) – The name of the token to get child tokens for.

Returns

The child tokens for the given token. If the given token is not a parent to any child tokens, return nothing.

Return type list[str]

get_mapping_details ()

Get the parse-mapping details of our entire Context.

Basically, we take the collection of all of the mapping details of the Context, as is, which we know is the "sum" of all of the Context's plugin's mapping_details. But we also yield each plugin individually, in case some information was lost, along the way.

Yields *dict[str]* – The mapping details of this Context.

get_required_tokens ()

list[str]: Get the tokens for this Context that must be filled.

get_str (*resolve_with="*, *depth=-1*, *holdout=None*, *groups=None*, *display_tokens=False*)

Create a string of the Context's mapping.

Note: holdout and groups cannot have any common token names.

Parameters

- **resolve_with** (`iterable[str] or str`, optional) – The types of ways that our parser is allowed to resolve a path. These are typically some combination of ('glob', 'regex', 'env') are are defined with our Plugin objects that make up a Context.
- **depth** (`int`, optional) – The number of times this method are allowed to expand the mapping before it must return it. If depth=-1, this method will expand mapping until there are no more tokens left to expand or no more subtokens to expand with. Default: -1.
- **holdout** (`set[str]`, optional) – If tokens (pre-existing or expanded) are in this list, they will not be resolved. Default is None.
- **groups** (`dict[str, str] or iterable[str, str]`, optional) – A mapping of token names and a preferred token value to substitute it with. This variable takes priority over all types of resolve_with types. Default is None.
- **display_tokens** (`bool`, optional) – If True, the original name of the token will be included in the output of the mapping, even it its contents are expanded. Example: '/some/{JOB}/here' -> r'/some/(?P<JOB>w+_d+)/here'. It's recommended to keep this variable as False because the syntax used is only regex-friendly. But if you really want it, it's there. Default is False.

Raises

- `ValueError` – If groups got a bad value.
- `ValueError` – If groups and holdout have any common elements. It's impossible to know what to do in that case because both items have conflicting instructions.

Returns The resolved string.

Return type str

`get_token_parse`(*name*, *parse_type*)
Get the parse expression for some token name.

Parameters

- **`name`** (`str`) – The name of the token to get parse details from.
- **`parse_type`** (`str`) – The engine type whose expression will be returned

Returns The parse expression used for the given token.

`get_tokens`(*required_only=False*)
Get the tokens in this instance.

Parameters **`required_only`** (`bool`, optional) – If True, do not return optional tokens. If False, return all tokens, required and optional. Default is False.

Returns The requested tokens.

Return type list[str]

`get_value_from_parent`(*name*, *parent*, *parse_type*)
Get the value of a token using another parent token.

Parameters

- **`name`** (`str`) – The token to get the value of.
- **`parent`** (`str`) – The parent token that is believed to have a value. If it has a value, it is used to parse and return a value for the name token.
- **`parse_type`** (`str`) – The parse engine to use.

Returns The value of the name token.

`is_valid`(*token*, *value*, *resolve_with='regex'*)
Check if a given value will work for some Ways token.

Parameters

- **`token`** (`str`) – The token to use to check for the given value.
- **`value`** – The object to check for validity.
- **`resolve_with`** (`str`, optional) – The parse type to use to check if value is valid for token. Only 'regex' is supported right now. Default: 'regex'.

Returns If the given value was valid.

Return type bool

`classmethod resolve_with_tokens`(*mapping*, *tokens*, *details*, *options*, *groups*, *display_tokens*)
Substitute tokens in our mapping for anything that we can find.

Parameters

- **`mapping`** (`str`) – The path that will be resolved and substituted.

- **tokens** (*list[str]*) – The pieces inside of the mapping to resolve.
- **details** (*dict[str]*) – All of the token/subtoken information available to resolve the tokens in this mapping.
- **options** (*list[str]*) – The different ways to resolve the tokens.
- **groups** (dict[str, str] or iterable[str, str], optional) – A mapping of token names and a preferred token value to substitute it with. This variable takes priority over all types of resolve_with types. Default is None.
- **display_tokens** (bool, optional) – Whether or not to add regex (?P<TOKEN_NAME>) tags around all of our resolved text.

Returns The resolved mapping.

Return type str

`ways.api.`**`add_descriptor`**(*description*, *update=True*)

Add an object that describes the location of Plugin objects.

Parameters

- **description** (*dict or str*) – Some information to create a descriptor object from. If the descriptor is a string and it is a directory on the path, ways.api.FolderDescriptor is returned. If it is an encoded URI, the string is parsed into a dict and processed. If it's a dict, the dictionary is used, as-is.
- **update** (bool, optional) – If True, add this Descriptor's plugins to Ways. If False, the user must register a Descriptor's plugins. Default is True.

`ways.api.`**`add_search_path`**(*description*, *update=True*)

Add an object that describes the location of Plugin objects.

Parameters

- **description** (*dict or str*) – Some information to create a descriptor object from. If the descriptor is a string and it is a directory on the path, ways.api.FolderDescriptor is returned. If it is an encoded URI, the string is parsed into a dict and processed. If it's a dict, the dictionary is used, as-is.
- **update** (bool, optional) – If True, add this Descriptor's plugins to Ways. If False, the user must register a Descriptor's plugins. Default is True.

`ways.api.`**`add_plugin`**(*path*)

Load the Python file as a plugin.

Parameters path (*str*) – The absolute path to a valid Python file (py or pyc).

`ways.api.`**`get_all_plugins`**()

list[*ways.api.Plugin*]: Every registered plugin.

`ways.api.`**`init_plugins`**()

Create the Descriptor and Plugin objects found in our environment.

This method ideally should only ever be run once, when Ways first starts.

class `ways.api.`**`Find`**(*context*)

Bases: *ways.core.compat.DirMixIn*, object

A wrapper around a Context object that provides some basic syntax-sugar.

The syntax of using Context objects is clunky. This class is meant to help. See the second and last example, for details.

Example

```
>>> context = Context('/some/context')
>>> command = context.get_action('get_assets')
>>> command()
>>> # ['/some/asset1.tif', '/some/asset2.tif', '/some/asset2.tif']
```

Example

```
>>> # If an action is meant to return back an iterable object and the
>>> # action that it gets back is None, that can cause immediate problems
>>> #
>>> context = Context('/some/context')
>>> command = context.get_action('get_assets')  # Returns None
>>> for asset in command():
>>>     print(asset)
>>> # The above code will TypeError error if get_action returns None
```

Example

```
>>> # The best you can do is this
>>> context = Context('/some/context')
>>> command = context.get_action('get_assets') or lambda: []
>>> for asset in command():
>>>     print(asset)
>>> # The above code will not error but it's pretty verbose compared to
>>> # what we're actually trying to accomplish.
```

Example

```
>>> # Here is (IMO) the best solution
>>> context = Context('/some/context')
>>> find = finder.Find(context)
>>> # Returns [] even if get_assets isn't defined
>>> # because get_assets is listed in Finder(context).defaults
>>> #
>>> for asset in find.get_assets():
>>>     print(asset)
```

classmethod add_to_defaults(*name*, *value*, *hierarchy=None*)

Add default value if an Action name is missing.

Parameters

- **name** (`str`) – The name of the Action to add a default value for.
- **value** – The object to add as the default return value for a missing Action.
- **hierarchy** (`tuple[str] or str`, optional) – The location to put these default values. If no hierarchy is given, ('default',) is used, instead.

classmethod clear()

Remove all stored default values from this class.

defaults = defaultdict(None, {('default',): {}})

class ways.api.**AssetFinder**(*finder*, *asset*)

Bases: *ways.core.compat.DirMixIn*, `object`

A class that wraps a Find class with the current asset.

Ways Action objects don't assume anything about their input. This is normally a good thing because it keeps Actions flexible. But if we're working with an Action that expects an Asset object, we'd have to do this all the time:

Example

```
>>> asset = resource.get_asset({'info': 'here'}, context='some/context')
>>> output = asset.context.actions.get_foo(action, some='other', args=4)
```

Gross, right?

So instead what we do is add AssetFinder as an 'actions' property and then forcefully pass the Asset as the first argument to Actions.

Example

```
>>> asset = resource.get_asset({'info': 'here'}, context='some/context')
>>> output = asset.actions.get_foo(some='other', args=4)
```

That's much better.

4.2.2 Inner Modules

ways.base.cache module

A set of functions to register objects to Ways.

ways.base.cache.**add_action**(*action*, *name=''*, *context=''*, *assignment='master'*)

Add a created action to Ways.

Parameters

- **action** (*ways.api.Action*) – The action to add. Action objects are objects that act on Context objects to gather some kind of information.
- **name** (`str`, optional) – A name to identify this action. The name must be unique to this hierarchy/assignment or it might override another pre-existing Action in the same location. If no name is given, the name on the action is tried, instead. Default: ''.
- **context** (*ways.api.Context* or str) – The Context or hierarchy of a Context to add this Action to.
- **assignment** (`str`, optional) – The group to add this action to, Default: 'master'.

Raises

- `RuntimeError` – If no hierarchy is given and no hierarchy could be found on the given action.
- `RuntimeError` – If no name is given and no name could be found on the given action.

- `ValueError` – If a Context object was given and no hierarchy could be found.

`ways.base.cache.`**`add_descriptor`**(*description*, *update=True*)

Add an object that describes the location of Plugin objects.

Parameters

- **`description`** (*`dict or str`*) – Some information to create a descriptor object from. If the descriptor is a string and it is a directory on the path, ways.api.FolderDescriptor is returned. If it is an encoded URI, the string is parsed into a dict and processed. If it's a dict, the dictionary is used, as-is.
- **`update`** (`bool`, optional) – If True, add this Descriptor's plugins to Ways. If False, the user must register a Descriptor's plugins. Default is True.

`ways.base.cache.`**`add_plugin`**(*path*)

Load the Python file as a plugin.

Parameters **`path`** (*`str`*) – The absolute path to a valid Python file (py or pyc).

`ways.base.cache.`**`add_search_path`**(*description*, *update=True*)

Add an object that describes the location of Plugin objects.

Parameters

- **`description`** (*`dict or str`*) – Some information to create a descriptor object from. If the descriptor is a string and it is a directory on the path, ways.api.FolderDescriptor is returned. If it is an encoded URI, the string is parsed into a dict and processed. If it's a dict, the dictionary is used, as-is.
- **`update`** (`bool`, optional) – If True, add this Descriptor's plugins to Ways. If False, the user must register a Descriptor's plugins. Default is True.

`ways.base.cache.`**`get_all_plugins`**()

list[*`ways.api.Plugin`*]: Every registered plugin.

`ways.base.cache.`**`get_assignments`**(*hierarchy*)

list[str]: Get the assignments for a hierarchy key in plugins.

`ways.base.cache.`**`init_plugins`**()

Create the Descriptor and Plugin objects found in our environment.

This method ideally should only ever be run once, when Ways first starts.

`ways.base.cache.`**`resolve_descriptor`**(*description*)

Build a descriptor object from different types of user input.

Parameters **`description`** (*`dict or str`*) – Some information to create a descriptor object from. If the descriptor is a string and it is a directory on the path, ways.api.Descriptor is returned. If it is an encoded URI, the string is parsed into a dict and processed. If it's a dict, the dictionary is used, as-is.

Returns Some descriptor object that works with the given input.

Return type `ways.api.Descriptor` or NoneType

`ways.base.cache.`**`update_plugins`**()

Look up every plugin in every descriptor and register them to Ways.

ways.base.commander module

A set of classes and functions used to extend a Context's interface.

Action objects attach to Contexts and let you change data on the Context or to run any kind of function.

ways.base.commander.**Action**
alias of `_Aktion`

class ways.base.commander.**ActionRegistry**
Bases: `type`

A metaclass that adds new Action objects to a registry-cache.

ways.base.commander.**add_action**(*action*, *name=''*, *context=''*, *assignment='master'*)
Add a created action to this cache.

Parameters

- **action** (*ways.api.Action*) – The action to add. Action objects are objects that get passed a Context object and run a function.
- **name** (`str`, optional) – A name to use with this action. The name must be unique to this hierarchy/assignment or it risks overriding another Action that might already exist at the same location. If no name is given, the name on the action is tried, instead.
- **context** (*ways.api.Context* or str) – The Context or hierarchy of a Context to add this Action to.
- **assignment** (`str`, optional) – The group to add this action to, Default: 'master'.

Raises

- `RuntimeError` – If no hierarchy is given and no hierarchy could be found on the given action.
- `RuntimeError` – If no name is given and no name could be found on the given action.

ways.base.connection module

A module that has a strategies for resolving Context/Plugin conflicts.

Depending on the Context object's attributes, it may be best to return a compound of all of the Context object's plugins, or the first-defined one or maybe the last defined Plugin object's value or even some other, special behavior.

The point is, whatever the strategy is, this module contains all of the different ways that a Context object's Plugin's values 'resolve' into a single output.

Note: In all cases, the plugins that are given to these functions are assumed to be in 'ascending' order. In other words, the 0th index of plugins is the oldest plugin and the -1 index is the latest plugin.

ways.base.connection.**generic_iadd**(*obj*, *other*)
Unify the different ways that built-in Python objects implement iadd.

It's important to note that this method is very generic and also unfinished. Feel free to add any other others, as needed. As long as they return a non-None value when successful and a None value when unsuccessful, any method is okay to use. (The logic for the non-None/None can be changed, too).

Parameters

- **obj** – Some object to add.
- **other** – An object to add into obj.

Returns The value that obj removes once other is added into it.

`ways.base.connection.`**`get_intersection_priority`**(*plugins*, *method*)

Get only the common elements from all plugin Objects and return them.

Note: Right now this function is only needed for get_groups() but could be abstracted if necessary, later.

Parameters **`plugins`** (list[*`ways.api.Plugin`*]) – The plugins to get the the intersected values from.

Returns The intersected value from all of the given Plugin objects.

`ways.base.connection.`**`get_left_right_priority`**(*plugins*, *method*)

Add all values of all plugins going from start to finish (left to right).

Parameters

- **`plugins`** (list[*`ways.api.Plugin`*]) – The plugins to get the the values from.
- **`method`** (callable[*`ways.api.Plugin`*]) – The callable function to get some value from a Plugin object.

Returns The compound value that was created from all of the plugins.

`ways.base.connection.`**`get_right_most_priority`**(*plugins*, *method*)

Get the most-latest value of the given plugins.

Note: If a Plugin runs method() successfully but gives a value that returns False (like '', or dict(), or [], etc), keep searching until an explicit value is found.

Parameters

- **`plugins`** (list[*`ways.api.Plugin`*]) – The plugins to get the the values from.
- **`method`** (callable[*`ways.api.Plugin`*]) – The callable function to use to call some value from a Plugin object.

Raises `NotImplementedError` – If the given method has no implementation in all of the given Plugin objects or if the output value would have been None.

Returns The output type of the given method.

`ways.base.connection.`**`try_and_return`**(*methods*)

Try every given method until one of them passes and returns some value.

Parameters **`methods`** (*`iterable[callable]`*) – Functions that takes no arguments to run.

Returns The output of the first method to execute successfully or None.

ways.base.descriptor module

A module that holds classes which abstract how Plugin objects are created.

A descriptor string could be a path to a file or folder or even to a database

`class` `ways.base.descriptor.`**`FileDescriptor`**(*items*)

Bases: `object`

A generic class that creates Plugin objects from a file, on-disk.

Note: Any FileDescriptor class that returns back Plugin objects is valid (Descriptors can query from a database, locally on file, etc, etc. It's all good) except there is one major requirement. FileDescriptor-like objects cannot append asynchronously. In other words, If a FileDescriptor manages threads that each find Plugin objects and append to a list of Plugin objects whenever each thread finishes, the Plugin objects might append out of order - which will create results that will be hard to debug.

It's recommended to not use threading at all unless this return process is managed (like with a queue or some other kind of idea).

classmethod filter_plugin_files(*items*)
Only get back the files that are likely to be plugin files.

get_plugin_info(*path*)
Given some file path, get its metadata info.

Parameters **path** (*str*) – The path to some directory containing plugin objects.

Returns The information about this plugin path.

Return type dict[str]

get_plugins(*items=None*)
Get the Plugin objects that this instance is able to find.

Note: This object will sort the items it is given before retrieving its Plugin objects so files/folders that are given different priorities depending on how their paths are named.

Parameters **items** (*iterable[str] or str*) – The paths that this FileDescriptor looks for to find Plugin objects. If not items are given, the instance's stored items are used, instead.

Returns The plugins.

Return type list[*ways.api.Plugin*]

classmethod get_supported_extensions()
list[str]: The Plugin file extensions.

class ways.base.descriptor.**FolderDescriptor**(*items*)
Bases: *ways.base.descriptor.FileDescriptor*

A generic abstraction that helps find Plugin objects for our code.

Note: Any FileDescriptor class that returns back Plugin objects is valid (Descriptors can query from a database, locally on file, etc, etc. It's all good) except there is one major requirement. FileDescriptor-like objects cannot append asynchronously. In other words, If a FileDescriptor manages threads that each find Plugin objects and append to a list of Plugin objects whenever each thread finishes, the Plugin objects might append out of order - which will create results that will be hard to debug.

It's recommended to not use threading at all unless this return process is managed (like with a queue or some other kind of idea).

class ways.base.descriptor.**GitLocalDescriptor**(*path*, *items*, *branch='master'*)
Bases: *ways.base.descriptor.FolderDescriptor*

An object to describe a local git repository.

This class conforms its input to files that its base class can read and then calls it. Otherwise that, it's not special.

class ways.base.descriptor.**GitRemoteDescriptor**(*url*, *items*, *path=''*, *branch='master'*)
Bases: *ways.base.descriptor.GitLocalDescriptor*

A Descriptor that clones an online Git repository.

ways.base.descriptor.**find_loader**(*path*)
Get the callable method needed to parse this file.

Parameters **path** (*str*) – The path to get the loader of.

Returns

A method that is used to load a Python file object for the given path.

Return type callable[file]

ways.base.descriptor.**is_invalid_plugin**(*hierarchy*, *info*)
Detect if a plugin's hierarchy is invalid, given its loaded information.

A plugin that has cyclic dependencies is considered "invalid".

Example

```
>>> cat plugin.yml
... plugins:
...     relative_plugin1:
...         hierarchy: mocap
...         mapping: '{root}/scenes/mocap'
...         uses:
...             - mocap
```

In the above example, that plugin refers to itself and could cause runtime errors.

Parameters

- **hierarchy** (*str*) – Some hierarchy to check.
- **info** (*dict[str]*) – The loaded information of some plugin from a Plugin Sheet file.

Returns If the plugin is valid.

ways.base.descriptor.**try_load**(*path*, *default=None*)
Try our best to load the given file, using a number of different methods.

The path is assumed to be a file that is serialized, like JSON or YAML.

Parameters

- **path** (*str*) – The absolute path to some file with serialized data.
- **default** (dict, optional) – The information to return back if no data could be found. Default is an empty dict.

Returns The information stored on this object.

Return type dict

ways.base.factory module

The main class that is used to create and store Context instances.

This setup is what makes ways.api.Context objects into flyweight objects.

Todo: Remove this class. It could just be a list with functions.

class `ways.base.factory.`**`AliasAssignmentFactory`**(*class_type*)

Bases: `ways.base.factory._AssignmentFactory`

Extend the _AssignmentFactory object to include Context aliases.

`clear`()

Remove all the stored aliases in this instance.

`get_instance`(*hierarchy*, *assignment*, *follow_alias=False*, *force=False*)

Get an instance of our class if it exists and make it if does not.

Parameters

- **`hierarchy`** (*`tuple[str] or str`*) – The location to look for our instance.
- **`assignment`** (*`str`*) – The category/grouping of the instance.
- **`follow_alias`** (`bool`, optional) – If True, the instance's hierarchy is assumed to be an alias for another hierarchy and the returned instance will use the "real" hierarchy. If False, the instance will stay as the aliased hierarchy, completely unmodified. Default is False.
- **`force`** (`bool`, optional) – If False and the Context has no plugins, return None. If True, an empty Context is returned. Default is False.

Returns An instance of our preferred class. If the Context that is called does not have any Plugin objects defined for it, it's considered 'empty'. To avoid problems in our code later, we return None, by default unless force is True.

Return type self._class_type() or NoneType

`is_aliased`(*hierarchy*)

bool: If this hierarchy is an alias for another hierarchy.

`resolve_alias`(*hierarchy*)

Assuming that the given hierarchy is an alias, follow the alias.

Parameters **`hierarchy`** (*`tuple[str] or str`*) – The location to look for our instance. In this method, hierarchy is expected to be an alias for another hierarchy so we look for the real hierarchy, here.

Returns The base hierarchy that this alias is meant to represent.

Return type tuple[str]

ways.base.finder module

The main class/functions used to find actions for Context/Asset objects.

class `ways.base.finder.`**`Find`**(*context*)

Bases: *`ways.core.compat.DirMixIn`*, `object`

A wrapper around a Context object that provides some basic syntax-sugar.

The syntax of using Context objects is clunky. This class is meant to help. See the second and last example, for details.

Example

```
>>> context = Context('/some/context')
>>> command = context.get_action('get_assets')
>>> command()
>>> # ['/some/asset1.tif', '/some/asset2.tif', '/some/asset2.tif']
```

Example

```
>>> # If an action is meant to return back an iterable object and the
>>> # action that it gets back is None, that can cause immediate problems
>>> #
>>> context = Context('/some/context')
>>> command = context.get_action('get_assets')  # Returns None
>>> for asset in command():
>>>     print(asset)
>>> # The above code will TypeError error if get_action returns None
```

Example

```
>>> # The best you can do is this
>>> context = Context('/some/context')
>>> command = context.get_action('get_assets') or lambda: []
>>> for asset in command():
>>>     print(asset)
>>> # The above code will not error but it's pretty verbose compared to
>>> # what we're actually trying to accomplish.
```

Example

```
>>> # Here is (IMO) the best solution
>>> context = Context('/some/context')
>>> find = finder.Find(context)
>>> # Returns [] even if get_assets isn't defined
>>> # because get_assets is listed in Finder(context).defaults
>>> #
>>> for asset in find.get_assets():
>>>     print(asset)
```

classmethod add_to_defaults(*name*, *value*, *hierarchy=None*)

Add default value if an Action name is missing.

Parameters

- **name** (*str*) – The name of the Action to add a default value for.
- **value** – The object to add as the default return value for a missing Action.

- **hierarchy** (`tuple[str] or str`, optional) – The location to put these default values. If no hierarchy is given, ('default',) is used, instead.

classmethod clear()
Remove all stored default values from this class.

defaults = defaultdict(None, {('default',): {}})

ways.base.plugin module

A module that holds Plugin classes and objects that combine into a Context.

class `ways.base.plugin.`**DataPlugin**(*name*, *sources*, *info*, *assignment*)
Bases: *ways.base.plugin.Plugin*

An add-on that was made from a serialized file (JSON/YAML/etc).

This class behaves exactly like a regular Plugin object and is stored in the same space as Plugin objects.

DataPlugin does not add itself to the cache automatically. It is the responsibility of some other class/function to register it to Ways.

We do this so that we can have better control over the DataPlugin's args and its assignment before it hits the cache.

add_to_registry = False

get_assignment()
str: Where this Plugin lives in Ways, along with its hierarchy.

get_groups()
Get the groups that this Plugin evaluates onto.

Note: The term 'groups' is not the same as the assignment of a Plugin. They are two different things.

Returns The groups.

Return type tuple[str]

get_hierarchy()
tuple[str] or str: The location that this Plugin exists within.

get_mapping()
str: The physical location of this Plugin (on the filesystem).

get_mapping_details()
dict[str]: Information about the mapping, if needed.

get_max_folder()
str: The furthest location up that this plugin can navigate to.

get_platforms()
set[str]: The platforms that this Plugin is allowed to run on.

get_uses()
tuple[str]: The Context hierarchies this instance depends on.

get_uuid()
str: A unique ID for this plugin.

is_path()
If the mapping is a filepath or None if unsure.

Returns If the mapping is a path to a file/folder on disk.

Return type bool or NoneType

class ways.base.plugin.**Plugin**
Bases: `object`

An add-on that is later retrieved by Context to gather its data.

add_to_registry = True

data
**dict[str]* – The display properties (like {'color'* – 'red'}).

class ways.base.plugin.**PluginRegistry**
Bases: `type`

A metaclass that adds new Plugin objects to a cache.

ways.base.plugin.**get_assignment**(*obj*)
str: Get an object's assignment or fallback to ways.DEFAULT_ASSIGNMENT.

ways.base.situation module

The most important part of Ways - A module that contains Context objects.

Context objects are persistent objects that are created using a Flyweight pattern. Once one instance of a Context is created, it is reused whenever the same Context is called again - which means it can be used like a monostate.

Reference: http://sourcemaking.com/design_patterns/flyweight http://sourcemaking.com/design_patterns/flyweight/python/1

The Python example is written in Python 3 but is the same idea.

Parts of the Context are generated at runtime and cannot be directly modified (like, for example, its Plugin objects). Other parts are dynamic (like the Context.data property).

class ways.base.situation.**Context**(*hierarchy*, *assignment="*, *connection=None*)
Bases: `object`

A collection of plugins that are read in order to resolve its methods.

as_dict(*changes=True*)
Convert this object into a dictionary.

This is different from a standard repr(Context) because it will include items that are not part of the Context's initialization.

It also creates a deepcopy of its contents, so that any changes to this dictionary won't affect the original object.

Parameters **changes** (`bool`, optional) – If True, the output will contain original plugin data as well as any changes that the user made over top of the original. If False, only the original information is returned. Default is True.

Returns A copy of the current information of this class.

Return type dict[str]

checkout(*assignment='master'*)
Make a new Context instance and return it, with the same hierarchy.

Parameters **`assignment`** (`str`, optional) – The new assignment to get.

Returns The new Context object.

Return type `ways.api.Context`

`data`
dict[str] – Data that was automatically generated and user data.

`get_action`(*name*)
`ways.api.Action` or callable or NoneType: The Action.

`get_all_plugins`(*hierarchy=''*, *assignment=''*)
list[`ways.api.Plugin`]: The found plugins, if any.

`get_all_tokens`()
Get the tokens in this Context's mapping and any subtokens.

Subtokens are tokens that are inside another token's mapping.

Returns All of the tokens known to this Context.

Return type set[str]

`get_assignment`()
str: The assignment for this Context.

`get_groups`()
tuple[str]: The groups that this Context belongs to.

`get_hierarchy`()
tuple[str]: The path to this Context.

`get_mapping`()
str: The mapping that describes this Context.

`get_mapping_details`()
Get the information that describes a Context instance's mapping.

This function is the same as "mapping_details" key that you'd see in a Plugin Sheet file and is critical to how a Context's parser builds into a file path.

Without it, you cannot get a proper filepath out of a Context.

Returns The information.

Return type dict[str]

`get_mapping_tokens`(*mapping=''*)
list[str]: Get all of the tokens that are in this Context's mapping.

`get_max_folder`()
str: The highest mapping point that this Context lives in.

`get_parser`()
`ways.api.ContextParser`: A parser copy that points to this Context.

`get_platforms`()
Get The OSes that this Context runs on.

The recognized platforms for this method is anything that platform.system() would return. (Examples: ['darwin', 'java', 'linux', 'windows']).

Returns The platforms that this Context is allowed to run on.

Return type set[str]

get_str(**args*, ***kwargs*)

Get the Context's mapping as filled-out text.

Parameters

- ***args** (*list*) – Positional args to send to ways.api.ContextParser.get_str.
- ****kwargs** (*dict*) – Keyword args to send to ways.api.ContextParser.get_str.

is_path()

bool: If the user indicated that the given mapping is a filepath.

plugins

Find all of the "valid" plugins for this instance.

What decides if a Plugin is "found" depends on a number of factors. First, the plugin needs to be inside the hierarchy of the Context, have the same assignment, and the platform(s) assigned to the Plugin must match our current system's platform. If a platform override is specified (aka if WAYS_PLATFORM has been set) then the Plugin object's platform has to match that, instead.

Raises

- `ValueError` – If the platform in WAYS_PLATFORM was invalid.
- `ValueError` – If the plugin found has a platform that is not found in our recognized_platforms variable.

Returns The found plugins.

Return type list[*ways.api.Plugin*]

revert()

Set the data on this instance back to its default.

classmethod validate_plugin(*plugin*)

Check if a plugin is "valid" for this Context.

Typically, a plugin is invalid if it was meant for a different OS (example), a Windows plugin shouldn't be added to a Context that is being run on a Linux machine.

Parameters plugin (*ways.api.Plugin*) – The plugin to check.

Raises

- `OSError` – If the user specified an unrecognized environment using the PLATFORM_ENV_VAR environment variable.
- `EnvironmentError` – If the plugin's environment does not match this environment.

Returns The plugin (completely unmodified).

Return type *ways.api.Plugin*

ways.base.situation.**clear_aliases**()

Remove all the stored aliases in this instance.

ways.base.situation.**clear_contexts**()

Remove every Context instance that this object knows about.

If a Context is re-queried after this method is run, a new instance for the Context will be created and returned.

Running this method is not recommended because it messes with the internals of Ways.

ways.base.situation.**get_all_contexts**()

Get or Create every Context instance that has plugins.

> **Warning:** This method can potentially be slow if there are a lot of Context objects left to be defined. That said, the second time this method is called, it'll be fast because the Context instances will be retrieved from the Context flyweight cache.

Returns Every Context object found by Ways.

Return type list[`ways.api.Context`]

`ways.base.situation.`**`get_context`**(*hierarchy*, *assignment=''*, *follow_alias=False*, *force=False*, **args*, ***kwargs*)

Get a persistent Context at some hierarchy/assignment location.

This function uses a Flyweight factory to manage the instance objects that it returns.

Reference: http://sourcemaking.com/design_patterns/flyweight http://sourcemaking.com/design_patterns/flyweight/python/1

Parameters

- **`hierarchy`** (`tuple[str]`) – The location to look for our instance.
- **`assignment`** (`str`) – The category/grouping of the instance. If no assignment is given, Ways will gather plugins in the order defined in the WAYS_PRIORITY environment variable and create plugins based on that.
- **`*args`** (`list`) – If an object instance is found at the hierarchy/assignment, this gets passed to the instantiation of that object.
- **`**kwargs`** (`dict[str]`) – If an object instance is found at the hierarchy/assignment, this gets passed to the instantiation of that object.

Returns An instance of that class. If the Context that is queried doesn't have any Plugin objects defined for it, it's considered 'empty'. To avoid faults in our code, we return None.

Return type <class_tuple instance> or NoneType

`ways.base.situation.`**`get_current_platform`**()

Get the user-defined platform for Ways.

If WAYS_PLATFORM is not defined, the user's system OS is returned instead.

Returns The platform.

Return type str

`ways.base.situation.`**`register_context_alias`**(*alias_hierarchy*, *old_hierarchy*)

Set a hierarchy to track the changes of another hierarchy.

This function lets you refer to plugins and Context objects without specifying their full names.

Example

```
>>> class SomePlugin(plugin.Plugin):
>>>     def get_mapping(self):
>>>         return '/some/path/here'
```

```
>>>     def get_hierarchy(self):
>>>         return ('nuke', 'scenes')
>>>     def get_platforms(self):
>>>         return '*'
```

```
>>> class AnotherPlugin(plugin.Plugin):
>>>     def get_mapping(self):
>>>         return '/some/path/here'
```

```
>>>     def get_hierarchy(self):
>>>         return ('maya', 'scenes')
```

```
>>>     def get_platforms(self):
>>>         return '*'
```

```
>>> sit.register_context_alias('maya_scenes', 'maya/scenes')
>>> context = ways.api.get_context('maya_scenes')
>>> # The resulting Context object has the hierarchy ('maya_scenes', )
>>> # but has all of the plugins from 'maya/scenes'
```

```
>>> sit.register_context_alias('maya_scenes', 'nuke/scenes')
```

```
>>> # Now, the Context('maya_scenes') is pointing to a Nuke Context
>>> # we can immediately work with this Context without having to
>>> # re-instantiate the Context
```

Raises `ValueError` – If the alias is the same as the hierarchy that it's trying to be aliased to or if the alias was already defined.

`ways.base.situation.`**`resolve_alias`**`(`*hierarchy*`)`
Get the real hierarchy that the given alias represents.

Parameters **`hierarchy`** (*`iter[str] or str`*) – The alias hierarchy to convert.

Returns The real hierarchy.

Return type tuple[str]

ways.helper.common module

A collection of functions that are used by modules in this package.

This module is not likely to change often.

`ways.helper.common.`**`conform_decode`**`(`*info*`)`
Make sure that 'create_using' returns a single string.

`ways.helper.common.`**`decode`**`(`*obj*`)`
dict[str]: Convert a URL-encoded string back into a dict.

`ways.helper.common.`**`encode`**`(`*obj*`)`
Make the given descriptor information into a standard URL encoding.

Parameters

- **`obj`** (*`dict[str]`*) – The Descriptor information to serialize.
- **`is normally something like`** (*`This`*) –

- **{'create_using'** – ways.api.FolderDescriptor}.

Returns The output encoding.

Return type str

ways.helper.common.**expand_string**(*format_string*, *obj*)

Split a string into a dict using a Python-format string.

Warning: Format-strings that have two tokens side-by-side are invalid. They must have at least some character between them. This format_string is invalid '{NAME}{ID}', this format_string is valid '{NAME}_{ID}'.

Example

```
>>> shot = 'NAME_010'
>>> format_string = '{SHOT}_{ID}'
>>> expand_string(format_string, shot)
... {'SHOT': 'NAME', 'ID': '010'}
```

Parameters

- **format_string** (*str*) – The Python-format style string to use to split it.
- **obj** (*str*) – The string to split out into a dict.

Raises ValueError – If the format_string given is invalid.

Returns The created dict from our obj string.

Return type dict

ways.helper.common.**get_platforms**(*obj*)

tuple[str]: The the platform(s) for the given object.

ways.helper.common.**get_python_files**(*item*)

Get the Python files at some file or directory.

Note: If the given item is a Python file, just return it.

Parameters **item** (*str*) – The absolute path to a file or folder.

Returns The Python files at the given location.

Return type list[str]

ways.helper.common.**import_object**(*name*)

Import a object of any kind, as long as it is on the PYTHONPATH.

Parameters **name** (*str*) – An import name (Example: 'ways.api.Plugin')

Raises ImportError – If some object down the name chain was not importable or if the entire name could not be found in the PYTHONPATH.

Returns The imported module, classobj, or callable function, or object.

ways.helper.common.**memoize**(*function*)

Create cache of values for a function.

ways.helper.common.**split_hierarchy**(*obj*, *as_type=<type 'tuple'>*)

Split a hierarchy into pieces, using the "/" character.

Parameters

- **obj** (*str or tuple[str]*) – The hierarchy to split.
- **as_type** (callable, optional) – The iterable type to return the hierarchy.

Returns The hierarchy, split into pieces.

Return type tuple[str]

ways.helper.common.**split_into_parts**(*obj*, *split*, *as_type=<type 'tuple'>*)

Split a string-like object into parts, using some split variable.

Example

```
>>> path = 'some/thing'
>>> split_into_parts(path, split='/')
... ('some', 'thing')
```

Parameters

- **obj** (*str or iterable*) – The object to split.
- **split** (*str*) – The character(s) to split obj by.
- **as_type** (callable[iterable[str]], optional) – The type to return from this function. Default: tuple.

Returns The split pieces.

Return type as_type[str]

ways.helper.dict_classes module

Extended dictionary classes.

class ways.helper.dict_classes.**ReadOnlyDict**(*data=None*, *settable=False*)

Bases: _abcoll.Mapping, object

A dictionary whose items can be set to read-only, if need be.

setdefault(*key*, *value*)

Add the key to this dictionary if it does not exist.

Parameters

- **key** – The key to set.
- **value** – The value to set on our key, if the key does not exist.

ways.helper.dict_classes.**recursive_default_dict**()

Create a recursive collection.defaultdict(dict).

ways.parsing.engine module

A collection of functions for the parse types in Ways.

Maybe in the future this module will be a place where users can "register" their own engines but, for right now, lets just K.I.S.S and assume people will want regex for over 90% of their needs.

`ways.parsing.engine.`**`get_token_parse`**(*name*, *parser*, *parse_type*)
Get the parse token for some token name, using a given parse_type.

Parameters

- **name** (*str*) – The token to get the token parse of.
- **parser** (*ways.api.ContextParser*) – The parser which presumably contains any information needed to retrieve a token parse value.
- **parse_type** (*str*) – The engine to use when getting our token parse information. Example: 'regex'.

Returns The token parse.

Return type str

`ways.parsing.engine.`**`get_token_parse_regex`**(*name*, *parser*, *groups=False*)
Get the parse token for some token name, using regex.

Parameters

- **name** (*str*) – The token to get the token parse of.
- **parser** (*ways.api.ContextParser*) – The parser which presumably contains any information needed to retrieve a token parse value.
- **groups** (`bool`, optional) – Whether or not to include (?P<{foo}>) around every value in the returned dict. Warning: Using this on a nested token can cause nested groups so it's not always recommended to enable this. Default is False.

Returns The token parse.

Return type str

`ways.parsing.engine.`**`get_value_from_parent`**(*name*, *parent*, *parser*, *parse_type*)
Use a token or its parent to get some stored value from a parser.

Parameters

- **name** (*str*) – The token to get the value of. If no value is found for this token, parent is used to parse and return a value.
- **parent** (*str*) – The token which is a parent of the name token. This parent should have a value or be able to get a value which we then parse and return.
- **parser** (*ways.api.ContextParser*) – The parser which presumably contains any information needed to retrieve the name token's value.
- **parse_type** (*str*) – The engine to use when getting our token parse information. Example: 'regex'.

Returns The value for the name token.

Return type str

`ways.parsing.engine.`**`get_value_from_parent_regex`**(*name*, *parent*, *parser*)
Do a Parent-Search using regex and return its value.

Parameters

- **name** (*str*) – The token to get the value of. If no value is found for this token, parent is used to parse and return a value.
- **parent** (*str*) – The token which is a parent of the name token. This parent should have a value or be able to get a value which we then parse and return.
- **parser** (*ways.api.ContextParser*) – The parser which presumably contains any information needed to retrieve the name token's value.

Returns The value for the name token.

Return type str

ways.parsing.parse module

A module that holds ContextParser - A class fills in Context's mapping.

class ways.parsing.parse.**ContextParser**(*context*)

Bases: object

A class that's used to fill out missing values in a Context's mapping.

Some quick terms: If you see the word 'token', it means a piece of a string that needs to be filled out, such as 'some_{TOKEN}_here'.

A field is the token + its information. For example, if the token being filled out is optional or if input to a token is valid.

This class is meant to expand and resolve the tokens inside the mapping of a Context object.

get_all_mapping_details()

Get the combined mapping details of this Context.

Note: The "true" combined mapping details of our Context is actually just Context.get_mapping_details(). This method can produce different results because it is yielding/updating Context.get_mapping_details() with all of its plugin's data. Use with caution.

Returns The combined mapping_details of our Context and plugins.

Return type dict[str]

get_child_tokens(*token*)

Find the child tokens of a given token.

Parameters **token** (*str*) – The name of the token to get child tokens for.

Returns

The child tokens for the given token. If the given token is not a parent to any child tokens, return nothing.

Return type list[str]

get_mapping_details()

Get the parse-mapping details of our entire Context.

Basically, we take the collection of all of the mapping details of the Context, as is, which we know is the "sum" of all of the Context's plugin's mapping_details. But we also yield each plugin individually, in case some information was lost, along the way.

Yields *dict[str]* – The mapping details of this Context.

get_required_tokens()
list[str]: Get the tokens for this Context that must be filled.

get_str(*resolve_with=”*, *depth=-1*, *holdout=None*, *groups=None*, *display_tokens=False*)
Create a string of the Context's mapping.

Note: holdout and groups cannot have any common token names.

Parameters

- **resolve_with** (`iterable[str] or str`, optional) – The types of ways that our parser is allowed to resolve a path. These are typically some combination of ('glob', 'regex', 'env') are are defined with our Plugin objects that make up a Context.
- **depth** (`int`, optional) – The number of times this method are allowed to expand the mapping before it must return it. If depth=-1, this method will expand mapping until there are no more tokens left to expand or no more subtokens to expand with. Default: -1.
- **holdout** (`set[str]`, optional) – If tokens (pre-existing or expanded) are in this list, they will not be resolved. Default is None.
- **groups** (`dict[str, str] or iterable[str, str]`, optional) – A mapping of token names and a preferred token value to substitute it with. This variable takes priority over all types of resolve_with types. Default is None.
- **display_tokens** (`bool`, optional) – If True, the original name of the token will be included in the output of the mapping, even it its contents are expanded. Example: '/some/{JOB}/here' -> r'/some/(?P<JOB>w+_d+)/here'. It's recommended to keep this variable as False because the syntax used is only regex-friendly. But if you really want it, it's there. Default is False.

Raises

- `ValueError` – If groups got a bad value.
- `ValueError` – If groups and holdout have any common elements. It's impossible to know what to do in that case because both items have conflicting instructions.

Returns The resolved string.

Return type str

get_token_parse(*name*, *parse_type*)
Get the parse expression for some token name.

Parameters

- **name** (*`str`*) – The name of the token to get parse details from.
- **parse_type** (*`str`*) – The engine type whose expression will be returned

Returns The parse expression used for the given token.

get_tokens(*required_only=False*)
Get the tokens in this instance.

Parameters **required_only** (`bool`, optional) – If True, do not return optional tokens. If False, return all tokens, required and optional. Default is False.

Returns The requested tokens.

Return type list[str]

get_value_from_parent(*name*, *parent*, *parse_type*)
Get the value of a token using another parent token.

Parameters

- **name** (*str*) – The token to get the value of.
- **parent** (*str*) – The parent token that is believed to have a value. If it has a value, it is used to parse and return a value for the name token.
- **parse_type** (*str*) – The parse engine to use.

Returns The value of the name token.

is_valid(*token*, *value*, *resolve_with='regex'*)
Check if a given value will work for some Ways token.

Parameters

- **token** (*str*) – The token to use to check for the given value.
- **value** – The object to check for validity.
- **resolve_with** (str, optional) – The parse type to use to check if value is valid for token. Only 'regex' is supported right now. Default: 'regex'.

Returns If the given value was valid.

Return type bool

classmethod resolve_with_tokens(*mapping*, *tokens*, *details*, *options*, *groups*, *display_tokens*)
Substitute tokens in our mapping for anything that we can find.

Parameters

- **mapping** (*str*) – The path that will be resolved and substituted.
- **tokens** (*list[str]*) – The pieces inside of the mapping to resolve.
- **details** (*dict[str]*) – All of the token/subtoken information available to resolve the tokens in this mapping.
- **options** (*list[str]*) – The different ways to resolve the tokens.
- **groups** (dict[str, str] or iterable[str, str], optional) – A mapping of token names and a preferred token value to substitute it with. This variable takes priority over all types of resolve_with types. Default is None.
- **display_tokens** (bool, optional) – Whether or not to add regex (?P<TOKEN_NAME>) tags around all of our resolved text.

Returns The resolved mapping.

Return type str

ways.parsing.parse.**expand_mapping**(*mapping*, *details*)
Split the tokens in a mapping into subtokens, if any are available.

Parameters

- **mapping** (*str*) – The mapping to expand.
- **details** (*dict[str]*) – The information about the mapping that will be used to expand it.

Returns The expanded mapping.

Return type str

`ways.parsing.parse.`**`find_tokens`**`(`*mapping*`)`
list[str]: The tokens to fill in. inside of a mapping.

`ways.parsing.parse.`**`is_done`**`(`*mapping*`)`
bool: If there are still tokens to fill in, inside the mapping.

ways.parsing.registry module

Responsible for giving users the ability to swap Assets with other objects.

`ASSET_FACTORY (dict[tuple[str]`
dict[str]]: This dict should not be changed directly. You should use the functions in this module, instead.

It is a global dictionary that stores classes that are meant to swap for an Asset object. ASSET_FACTORY's key is the hierarchy of the Context and its value is another dict, which looks like this:

'class': The class to swap for. 'init': A custom inititialization function for the class (if needed). 'children': If True, the class is used for all hierarchies that build off of the given hierarchy. If False, the class is only added to the given hierarchy.

`ways.parsing.registry.`**`get_asset_class`**`(`*hierarchy*`)`
Get the class that is registered for a Context hierarchy.

`ways.parsing.registry.`**`get_asset_info`**`(`*hierarchy*`)`
Get the class and initialization function for a Context hierarchy.

Parameters **`hierarchy`** (*`tuple[str] or str`*) – The hierarchy to get the asset information of.

Returns The class type and the function that is used to instantiate it.

Return type tuple[classobj, callable]

`ways.parsing.registry.`**`make_default_init`**`(`*class_type*, **args*, ***kwargs*`)`
Just make the class type, normally.

`ways.parsing.registry.`**`register_asset_class`**`(`*class_type*, *context*, *init=None*, *children=False*`)`
Change get_asset to return a different class, instead of an Asset.

The Asset class is useful but it may be too basic for some people's purposes. If you have an existing class that you'd like to use with Ways,

Parameters

- **`class_type`** (*`classobj`*) – The new class to use, instead. context (str or *`ways.api.Context`*): The Context to apply our new class to.
- **`init`** (`callable`, optional) – A function that will be used to create an instance of class_type. This variable is useful if you need to customize your class_type's __init__ in a way that isn't normal (A common example: If you want to create a class_type that does not pass context into its __init__, you can use this variable to catch and handle that).
- **`children`** (`bool`, optional) – If True, this new class_type will be applied to child hierarchies as well as the given Context's hierarchy. If False, it will only be applied for this Context. Default is False.

`ways.parsing.registry.`**`reset_asset_classes`**`(`*hierarchies=()*`)`
Clear out the class(es) that is registered under a given hierarchy.

Parameters **hierarchies** (*iter[tuple[str]]*) – All of the hierarchies to remove custom Asset classes for. If nothing is given, all hierarchies will be cleared.

ways.parsing.resource module

Asset objects are objects that store per-instance data for Context objects.

They are necessary because Context objects are flyweights and, because of that, cannot carry instance data.

class `ways.parsing.resource.`**Asset**(*info*, *context*, *parse_type='regex'*)

Bases: `object`

An object that contains a Context and data about the Context.

The idea of this class is to keep Context information abstract, and let Context parse/use that information. Depending on what the Context is for, it could be used to ground the information to a filesystem or a database or some other structure that the Context knows about.

get_missing_required_tokens()

Find any token that still needs to be filled for our parser.

If a token is missing but it has child tokens and all of the child tokens are defined, it is excluded from the final output. If the missing token is a child of some parent token that is defined, then the value of the token is parsed. If the parse is successful, the token is excluded from the final output.

Returns Any tokens that have no value.

Return type list[str]

get_str(*required=True*, **args*, ***kwargs*)

Get the full path to the asset, if any.

Parameters

- **required** (`bool`, optional) – If True and there are tokens that are required that still are not filled, raise an error. If False, return the incomplete string. Default is True.
- ***args** (*list*) – Positional args to send to ContextParser.get_str.
- ****kwargs** (*list*) – Keywords args to send to ContextParser.get_str.

Raises `ValueError` – If required is True (in other words, we assume that)

Returns The resolved string for this instance.

Return type str

get_token_parse(*name*, *parse_type=''*)

Get the parse expression for some token name.

Parameters

- **name** (*str*) – The name of the token to get parse details from.
- **parse_type** (`str`, optional) – The engine type whose expression will be returned. If no parse_type is given, the stored parse_type is used.

Returns The parse expression used for the given token.

get_unfilled_tokens(*required_only=False*)

Get the tokens in this instance that still don't have values.

Parameters **required_only** (`bool`, optional) – If True, do not return optional tokens. If False, return all tokens, required and optional. Default is False.

Returns The tokens that still need values.

Return type list[str]

get_value(*name*, *real=False*)

Get some information about this asset, using a token-name.

If the information is directly available, we return it. If it isn't though, it is searched for, using whatever information that we do have.

If the token name is a child of another token that is defined, we use the parent token to "build" a value for the token that was requested.

If the token name is a parent of some other tokens that all have values, we try to "build" it again, by combining all of the child tokens.

In both cases, the return value is created but not defined. But it lets you do this:

Example

```
>>> shot_info = {
...     'JOB': 'someJob',
...     'SCENE': 'SOMETHING',
...     'SHOT': 'sh0010'  # Pretend SHOT_NUMBER is a child of SHOT
... }
>>> shot_asset = resource.Asset(shot_info, context='job/scene/shot')
>>> shot_asset.get_value('SHOT_NUMBER')
... # Result: '0010'
```

Parameters

- **name** (*str*) – The token to get the value of.
- **real** (bool, optional) – If True, the original parsed value is returned. If False and the given token has functions defined in "before_return" then those functions will process the output and then return it. Default is False.

Returns The value at the given token.

set_value(*key*, *value*, *force=False*)

Store the given value to some key.

Parameters

- **key** (*str*) – The token that our value will be stored into.
- **value** (*str*) – The value to store.
- **force** (bool, optional) – If False, values are checked against their tokens before being set. If True, values are set for each token, even if they are not valid input for that token. Default is False.

class ways.parsing.resource.**AssetFinder**(*finder*, *asset*)

Bases: *ways.core.compat.DirMixIn*, object

A class that wraps a Find class with the current asset.

Ways Action objects don't assume anything about their input. This is normally a good thing because it keeps Actions flexible. But if we're working with an Action that expects an Asset object, we'd have to do this all the time:

Example

```
>>> asset = resource.get_asset({'info': 'here'}, context='some/context')
>>> output = asset.context.actions.get_foo(action, some='other', args=4)
```

Gross, right?

So instead what we do is add AssetFinder as an 'actions' property and then forcefully pass the Asset as the first argument to Actions.

Example

```
>>> asset = resource.get_asset({'info': 'here'}, context='some/context')
>>> output = asset.actions.get_foo(some='other', args=4)
```

That's much better.

`ways.parsing.resource.`**`expand_info`**(*info*, *context=None*)

Get parsed information, using the given Context.

Note: This function requires regex in order to parse.

Todo: Maybe I can abstract the parser to use different parse options, like I did in get_value_from_parent. And then if that doesn't work, I can add the option to "register" a particular parser.

Parameters

- **info** (*dict[str] or str*) – The info to expand. If the input is a dict, it is passed through and returned. If it is a string, the string is parsed against the given context.
- **context** (<`ways.api.Context`, optional) – The Context that will be used to parse info. If no Context is given, the Context is automatically found. Default is None.

Raises `NotImplementedError` – If context is None. There's no auto-find-context option yet.

Returns The asset info.

Return type dict[str]

`ways.parsing.resource.`**`get_asset`**(*info*, *context=None*, **args*, ***kwargs*)

Get some class object that matches the given Context and wraps some info.

Parameters

- **info** (*dict[str] or str*) – The info to expand. If the input is a dict, it is passed through and returned. If it is a string, the string is parsed against the given context. Generally speaking, it's better to give a string that is an exact or partial match to a Context's mapping than it is to give a dict. This is doubly true if no context is given.
- **context** (*ways.api.Context* or str or tuple[str]', optional) – The Context to use for the asset. If a string is given, it is assumed to be the Context's hierarchy and a Context object is constructed. If nothing is given, the best possible Context is "found" and tried. This auto-find process will try to find the "best" match by looking at every known Context's mapping. A match is not guaranteed. Default is None.

- ***args** (*list*) – Optional position variables to pass to our found class's constructor.
- ****kwargs** (*dict*) – Optional keyword variables to pass to our found class's constructor.

Raises `NotImplementedError` – If context is None. There's no auto-find-context option yet.

Returns The found class object or NoneType. If no class definition was found for the given Context, return a generic Asset object.

ways.parsing.trace module

A module to help you debug all of your Context, Plugin, and Action objects.

`ways.parsing.trace.`**`get_action_hierarchies`**(*action*)

Get the Context hierachies that this Action is registered for.

Note: get_action_hierarchies will return every Action that matches the given Action name. So if multiple classes/functions are all registered under the same name, then every hierarchy that those Actions use will be returned. However, if a object like a function or class that was registered, only that object's hierarchies will be returned.

Parameters **action** (*str or class or callable*) – The action to get the hierachies of.

Returns The hierarchies for the given Action.

Return type set[tuple[str]]

`ways.parsing.trace.`**`get_all_action_hierarchies`**()

Organize every Action that is registered into Ways by object and hierarchy.

Returns

dict[str: str or set]]: Actions are stored as either classes or functions. Each Action's value is a dict which contains the hierachies that the Action is applied to and its registered name.

Return type dict[class or callable

`ways.parsing.trace.`**`get_all_assignments`**()

set[str]: All of the assignments found in our environment.

`ways.parsing.trace.`**`get_all_hierarchies`**()

set[tuple[str]]: The Contexts that have plugins in our environment.

`ways.parsing.trace.`**`get_all_hierarchy_trees`**(*full=False*)

Get a description of every Ways hierarchy.

Examples

```
>>> get_all_hierarchy_trees(full=True)
>>> {
>>>     ('foo', ): {
>>>         ('foo', 'bar'): {
>>>             ('foo' 'bar', 'fizz'): {},
>>>         },
>>>         ('foo', 'something', 'buzz'): {
>>>             ('foo', 'something', 'buzz', 'thing'): {},
>>>         },
>>>     },
>>> }
```

```
>>> get_all_hierarchy_trees(full=False)
>>> {
>>>     'foo': {
>>>         'bar': {
>>>             'fizz': {},
>>>         },
>>>         'something': {
>>>             'buzz': {
>>>                 'thing': {},
>>>             },
>>>         },
>>>     },
>>> }
```

Parameters **full** (bool, optional) – If True, each item in the dict will be its own hierarchy. If False, only a single part will be written. See examples for details. Default is False.

Returns The entire hierarchy.

Return type `collections.defaultdict[str]`

`ways.parsing.trace.`**`get_child_hierarchies`**`(`*hierarchy*`)`
list[tuple[str]]: Get hierarchies that depend on the given hierarchy.

`ways.parsing.trace.`**`get_child_hierarchy_tree`**`(`*hierarchy, full=False*`)`
Get all of the hierarchies that inherit the given hierarchy.

Examples

```
>>> get_all_hierarchy_trees(full=True)
>>> {
>>>     ('foo', ): {
>>>         ('foo', 'bar'): {
>>>             ('foo' 'bar', 'fizz'): {},
>>>         },
>>>         ('foo', 'something', 'buzz'): {
>>>             ('foo', 'something', 'buzz', 'thing'): {},
>>>         },
>>>     },
>>> }
```

```
>>> get_all_hierarchy_trees(full=False)
>>> {
>>>     'foo': {
>>>         'bar': {
>>>             'fizz': {},
>>>         },
>>>         'something': {
>>>             'buzz': {
>>>                 'thing': {},
>>>             },
>>>         },
>>>     },
>>> }
```

Parameters

- **hierarchy** (*tuple[str]*) – The hierarchy to get the child hierarchy items of.
- **full** (bool, optional) – If True, each item in the dict will be its own hierarchy. If False, only a single part will be written. See examples for details. Default is False.

Returns The entire hierarchy.

Return type `collections.defaultdict[str]`

`ways.parsing.trace.`**`startswith`**(*base*, *leaf*)
Check if all tuple items match the start of another tuple.

Raises `ValueError` – If base is shorted than leaf.

`ways.parsing.trace.`**`trace_action_names`**(*obj*, **args*, ***kwargs*)
Get the names of all actions available to a Ways object.

Parameters

- **obj** (*ways.api.Action* or *ways.api.AssetFinder* or *ways.api.Context* or *ways.api.Find*) – The object to get the action names of.
- ***args** (*list*) – Position args to pass to ways.get_action_names.
- ****kwargs** (*dict[str]*) – Keyword args to pass to ways.get_action_names.

Returns The names of all actions found for the Ways object.

Return type list[str]

`ways.parsing.trace.`**`trace_actions`**(*obj*, **args*, ***kwargs*)
Get actions that are assigned to the given object.

Parameters

- **obj** (*ways.api.Action* or *ways.api.AssetFinder* or *ways.api.Context* or *ways.api.Find*) – The object to get the actions of.
- ***args** (*list*) – Position args to pass to ways.get_actions.
- ****kwargs** (*dict[str]*) – Keyword args to pass to ways.get_actions.

Returns The actions in the hierarchy.

Return type list[*ways.api.Action* or callable]

`ways.parsing.trace.`**`trace_actions_table`**(*obj*, **args*, ***kwargs*)
Find the names and objects of every action registered to Ways.

Parameters

- **obj** (*ways.api.Action* or *ways.api.AssetFinder* or *ways.api.Context* or *ways.api.Find*) – The object to get the available actions table of.
- ***args** (*list*) – Position args to pass to ways.get_actions_info..
- ****kwargs** (*dict[str]*) – Keyword args to pass to ways.get_action_info.

Returns The names and actions of an object.

Return type dict[str, *ways.api.Action* or callable]

`ways.parsing.trace.`**`trace_all_descriptor_results`**`()`
list[dict[str]]: The load/failure information about each Descriptor.

`ways.parsing.trace.`**`trace_all_load_results`**`()`
Get the load results of every plugin and descriptor.

If the UUID for a Descriptor cannot be found, Ways will automatically assign it a UUID.

Using this function we can check 1. What plugins that Ways found and tried to load. 2. If our plugin loaded and, if not, why.

Returns The main dictionary has two keys, "descriptors" and "plugins". Each key has an OrderedDict that contains the UUID of each Descriptor and plugin and their objects.

Return type dict[str, `collections.OrderedDict` [str, dict[str]]]

`ways.parsing.trace.`**`trace_all_plugin_results`**`()`
list[dict[str]]: The results of each plugin's load results.

`ways.parsing.trace.`**`trace_assignment`**`(`*obj*`)`
str: Get the assignment for this object.

`ways.parsing.trace.`**`trace_context`**`(`*obj*`)`
Get a Context, using some object.

This function assumes that the given object is a Ways class that only has 1 Context added to it (not several).

Parameters **`obj`** – Some Ways object instance.

Returns The found Context.

Return type *`ways.api.Context`* or NoneType

`ways.parsing.trace.`**`trace_hierarchy`**`(`*obj*`)`
Try to find a hierarchy for the given object.

Parameters **`obj`** (*`ways.api.Action`* or *`ways.api.AssetFinder`* or *`ways.api.Context`* or *`ways.api.Find`*) – The object to get the hierarchy of.

Returns The hierarchy of some object.

Return type tuple[str]

ways.parsing.tracehelper module

Common functions used for tracing that cannot go into trace.py.

This module is used in situation.py. If a function in this module were used and imported by trace.py, it'd create a cyclic import.

`ways.parsing.tracehelper.`**`trace_method_resolution`**`(`*method*, *plugins=False*`)`
Show the progression of how a Context's method is resolved.

Parameters

- **`method`** (*`callable`*) – Some function on a Context object.
- **`plugins`** (`bool`, optional) – If False, the result at every step of the method will be returned. If True, the Plugin that created each result will be returned al along with the result at every step. Default is False.

Returns The plugin resolution at each step.

Return type list

4.2.3 Module contents

The main location where loaded plugin and action objects are managed.

`ways.`**`add_plugin`**(*plugin*, *assignment='master'*)
Add a plugin to Ways.

Parameters

- **`plugin`** (`ways.api.Plugin`) – The plugin to add.
- **`assignment`** (`str`, optional) – The assignment of the plugin. Default: 'master'.

`ways.`**`check_plugin_uuid`**(*info*)
Make sure that the plugin UUID is not already taken.

Parameters **`info`** (`ways.api.DataPlugin` or dict[str]) – Data that may become a proper plugin.

Raises `RuntimeError` – If the plugin's UUID is already taken.

`ways.`**`clear`**()
Remove all Ways plugins and actions.

`ways.`**`get_action`**(*name*, *hierarchy*, *assignment='master'*)
Find an action based on its name, hierarchy, and assignment.

The first action that is found for the hierarchy is returned.

Parameters

- **`name`** (*`str`*) – The name of the action to get. This name is assigned to the action when it is defined.
- **`hierarchy`** (*`tuple[str]`*) – The location of where this Action object is.
- **`assignment`** (`str`, optional) – The group that the Action was assigned to. Default: 'master'.

Returns The found Action object.

Return type `ways.api.Action` or NoneType

`ways.`**`get_action_names`**(*hierarchy*, *assignment='master'*)
Get the names of all actions available for some plugin hierarchy.

Parameters

- **`hierarchy`** (*`tuple[str]`*) – The specific description to get plugin/action objects from.
- **`assignment`** (`str`, optional) – The group to get items from. Default: 'master'.

Returns The names of all actions found for the Ways object.

Return type list[str]

`ways.`**`get_actions`**(*hierarchy*, *assignment='master'*, *duplicates=False*)
Get back all of the action objects for a plugin hierarchy.

Parameters

- **`hierarchy`** (*`tuple[str]`*) – The specific description to get plugin/action objects from.
- **`assignment`** (`str`, optional) – The group to get items from. Default: 'master'.

- **duplicates** (`bool`, optional) – If True, The first Action that is found will be returned. If False, all actions (including parent actions with the same name) are all returned. Default is False.

Returns The actions in the hierarchy.

Return type list[*`ways.api.Action`* or callable]

`ways.`**`get_actions_info`**(*hierarchy*, *assignment='master'*)

Get the names and objects for all Action objects in a hierarchy.

Parameters

- **hierarchy** (*`tuple[str]`*) – The specific description to get plugin/action objects from.
- **assignment** (`str`, optional) – The group to get items from. Default: 'master'.

Returns

`ways.api.Action` **or callable]:** The name of the action and its associated object.

Return type dict[str

`ways.`**`get_actions_iter`**(*hierarchy*, *assignment='master'*)

Get the actions at a particular hierarchy.

Parameters

- **hierarchy** (*`tuple[str]`*) – The location of where this Plugin object is.
- **assignment** (`str`, optional) – The group that the PLugin was assigned to. Default: 'master'. If assignment='', all plugins from every assignment is queried.

Yields dict[str, *`ways.api.Action`*] – The actions for some hierarchy.

`ways.`**`get_known_platfoms`**()

Find the platforms that Ways sees.

This will return back the platforms defined in the WAYS_PLATFORMS environment variable. If WAYS_PLATFORMS isn't defined, a default set of platforms is returned.

Returns

All of the platforms. Default: {'darwin', 'java', 'linux', 'windows'}

Return type set[str]

`ways.`**`get_parse_order`**()

list[str]: The order to try all of the parsers registered by the user.

`ways.`**`get_plugins`**(*hierarchy*, *assignment='master'*)

Find an plugin based on its name, hierarchy, and assignment.

Every plugin found at every level of the given hierarchy is collected and returned.

Parameters

- **name** (*`str`*) – The name of the plugin to get. This name needs to be assigned to the plugin when it is defined.
- **hierarchy** (*`tuple[str]`*) – The location of where this Plugin object is.
- **assignment** (`str`, optional) – The group that the PLugin was assigned to. Default: 'master'. If assignment='', all plugins from every assignment is queried.

Returns The found plugins, if any.

Return type list[*`ways.api.Plugin`*] `ways.`**`get_priority`**`()`

Determine the order that assignments are searched through for plugins.

This list is controlled by the WAYS_PRIORITY variable.

For example, os.environ['WAYS_PRIORITY'] = 'master:job'. Since job plugins come after master plugins, they are given higher priority

Todo: Give a recommendation (in docs) for where to read more about this.

Returns The assignments to search through.

Return type tuple[str]

4.2.4 Subpackages

ways.core package

Submodules

ways.core.check module

Comparison operators and other useful functions.

`ways.core.check.`**`force_itertype`**`(`*obj*, *allow_outliers=False*, *itertype=None*`)`

Change the given object into an iterable object, if it isn't one already.

Parameters

- **`obj`** (*`any`*) – The object(s) to wrap in a list iterable
- **`allow_outliers`** (`bool`, optional) – If True, returns True if obj is string
- **`itertype`** (*`callable`*) – Any iterable object that is callable, such as list, set, dict, etc.

Returns A list, containing objects if is_itertype is False

Return type list[obj]

`ways.core.check.`**`is_itertype`**`(`*obj*, *allow_outliers=False*, *outlier_check=<function is_string_instance>*`)`

Check if the obj is iterable. Returns False if string by default.

Parameters

- **`obj`** (*`any`*) – The object to check for iterable methods
- **`allow_outliers`** (`bool`, optional) – If True, returns True if obj is string
- **`outlier_check`** (`function`, optional) – A function to use to check for 'bad itertypes'. This function does nothing if allow_outliers is True. If nothing is provided, strings are checked and rejected.

Returns If the input obj is a proper iterable type.

Return type bool

`ways.core.check.`**`is_string_instance`**`(`*obj*`)`

bool: If the object is a string instance.

`ways.core.check.`**`is_string_type`**(*obj*)
bool: If the object is a string type.

ways.core.compat module

Python 2/3 compatibility classes and functions.

Yes, import six is awesome. This just covers whatever six doesn't.

class `ways.core.compat.`**`DirMixIn`**
Bases: `object`

'Mix-in to make implementing __dir__ method in subclasses simpler.

In Python 2, you can't call super() in __dir__. This mixin lets you do it.

Example

```
>>> class Something(object):
>>>     def __dir__(self):
>>>         return super(Something, self).__dir__()
```

```
>>> print(dir(Something()))   # Raises AttributeError
```

```
>>> class Something(DirMixIn, object):
>>>     def __dir__(self):
>>>         return super(Something, self).__dir__()
```

```
>>> print(dir(Something()))   # Works
```

That's all there is to it.

ways.core.grouping module

Various functions for grouping sequences of integers.

`ways.core.grouping.`**`chunkwise_iter`**(*seq*, *size=2*)
generator: Split the given sequence by *size*.

`ways.core.grouping.`**`filter_consecutive_items`**(*obj*)
Remove all consecutive elements but keep duplicate items.

Parameters **`obj`** (*`iterable`*) – The list (or iterable) to process

Returns The ranges from the given list

Return type list[int or tuple]

`ways.core.grouping.`**`get_difference`**(*list1*, *list2*)
Get the elements of list1 that are not in list2.

Note: This is NOT a symmetric_difference

Warning: This function will cause you to lose list order of list1 and list2

Parameters

- **list1** (*list*) – The list to get the intersection with
- **list2** (*list*) – The list to get the intersection against

Returns The combination of list1 and list2

Return type list

`ways.core.grouping.`**`get_ordered_intersection`**(*seq1*, *seq2*, *memory_efficient=False*)

Get the elements that exist in both given sequences.

This code will preserve the order of the first sequence given.

Parameters

- **seq1** (*iterable*) – The sequence to iterate. Also determines return order.
- **seq2** (*iterable*) – The second sequence to compare against the first
- **memory_efficient** (bool, optional) – If you know that every element in both sequences are small in size, enable this option for a potential speed boost.

Returns The common elements of the two sequences

Return type iterable[any]

`ways.core.grouping.`**`group_into`**(*seq*, *maximum*)

Break a sequence up in a specified number of groups.

Example

```
>>> seq = [1, 2, 3, 4, 5, 6, 7, 8, 9, 10]
>>> maximum = 4
>>> group_into(seq=seq, maximum=maximum)
>>> [[1, 4, 7, 10], [2, 5, 8], [3, 6, 9]]
```

Parameters

- **seq** (*iterable*) – The sequence to split up
- **maximum** (*int*) – The number of groups to make

Returns Group of the original iterable sequence object

Return type list[iterable]

`ways.core.grouping.`**`group_nth`**(*seq*, *by*)

Split the sequence by the given number.

Example

```
>>> seq = [1, 2, 3, 4, 5, 6, 7, 8, 9, 10]
>>> maximum = 4
>>> group_nth(seq=seq, by=by)
>>> [[1, 2, 3, 4], [5, 6, 7, 8], [9, 10]]
```

Note: The number of groups that will be made is (len(seq) // by) + 1

Parameters

- **seq** (*iterable*) – The sequence to split up into some groups
- **maximum** (*int*) – The size of the groups that will be made from the original seqence

Returns A group of the original iterable sequence object

Return type list[iterable]

`ways.core.grouping.`**`grouper`**(*obj*)

Group a list together by its items.

Slightly different result than ranges, in cases where outlier items lie between two ranges.

Example

```
>>> list(ranges([0, 1, 2, 3, 4, 7, 10, 12, 14, 16]))
>>> [xrange(0, 4, 1), xrange(10, 16, 2)]
```

Parameters **obj** (*iterable*) – The list (or iterable) to process

Returns The ranges from the given list

Return type list[int or tuple]

`ways.core.grouping.`**`has_common_elements`**(**args*)

bool: Tests a variable number of sequences for common elements.

`ways.core.grouping.`**`pairwise`**(*iterable*)

Change an iterable item in to pairs -> (s0,s1), (s1,s2), (s2, s3), ...

`ways.core.grouping.`**`ranges`**(*obj*, *return_range=True*)

Get the start and end ranges for a list of discontinuous int ranges.

Reference: http://stackoverflow.com/questions/39709606/

Example

```
>>> list(ranges([0, 1, 2, 3, 4  7, 10, 12, 14, 16]))
>>> [xrange(0, 4, 1), 7, xrange(10, 16, 2)]
```

Parameters

- **obj** (*list[int]*) – A list of integers to get the sequence of

- **return_range** (*bool*) – If you just need an iterable and you don't care about keeping the start/end/step, setting to True is more efficient on memory. If False, returns a tuple with start, end, and step.

Yields *[int or range or tuple]* – The ranges from the given list

ways.core.grouping.**uniquify_list**(*seq*, *idfun=None*)

Order preserving way to get unique elements in a list.

This function is a bit dirty but extremely fast (see benchmark).

Reference: https://www.peterbe.com/plog/uniqifiers-benchmark

Parameters

- **seq** (*list*) – The list to make unique
- **idfun** (*func*) – An optional function to run, as part of the uniquifier

Returns The uniquified list

Return type list

ways.core.loop module

For-loop helper methods.

ways.core.loop.**last_iter**(*iterable*)

Wrap a loop to determine when the last value of a loop is found.

Reference: https://stackoverflow.com/questions/1630320

Parameters **iterable** (*iterable*) – The objects to move through

ways.core.loop.**walk_items**(*obj*)

Iterate and yield parts of an object.

Example

```
>>> foo = ('fee', 'fi', 'fo', 'fum')
>>> print(list(walk_items(foo)))
>>> # Result: [('foo', ), ('foo', 'fi'), ('foo', 'fi', 'fo'),
>>> #           ('fee', 'fi', 'fo', 'fum')]
```

Note: This function requires the object use __len__, and __getitem__.

Parameters **obj** (*iterable*) – Some object to return the parts of.

Yields Parts of the object.

ways.core.pathrip module

Tools for dealing with file paths.

All of these functions should be OS-independent or at least indicate if not.

`ways.core.pathrip.`**`get_subfolder_root`**(*path*, *subfolders*, *method='tail'*)

Find the path of some path, using some consecutive subfolders.

Parameters

- **`path`** (*`str`*) – The path to get the root of.
- **`subfolders`** (*`list[str] or str`*) – The folders to search for in the path.
- **`method`** (`callable[str, list[str]] or str`, optional) – The strategy used to get the root subfolders. A function is acceptable or a preset string can be used. String options: 'tail': Search a path to get the longest possible match (the last tail).

Returns The root path that contains the subfolders.

Return type str

`ways.core.pathrip.`**`get_subfolder_root_tail`**(*path*, *subfolders*)

Get the longest match of some path, using some subfolders.

Example

```
>>> root = '/jobs/rnd_ftrack/shots/sh01/maya/scenes/modeling/RELEASE/some_file_
↪name.ma'
>>> subfolders = ['maya/scenes']
>>> get_subfolder_root_tail(root, subfolders)
'/jobs/rnd_ftrack/shots/sh01/maya/scenes'
```

Note: If a path has more than one match for subfolders, the last match is used.

Parameters

- **`path`** (*`str`*) – The path to get the root of.
- **`subfolders`** (*`list[str] or str`*) – The folders to search for in the path.

Returns The path that matches some subfolder or an empty string.

Return type str

`ways.core.pathrip.`**`split_os_path_asunder`**(*path*)

Split up a path, even if it is in Windows and has a letter drive.

Parameters **`path`** (*`str`*) – The path to split up into parts.

Returns The split path.

Return type list[str]

`ways.core.pathrip.`**`split_path_asunder`**(*path*)

Split a path up into individual folders.

This function is OS-independent but does not take into account Windows drive letters. For that, check out split_os_path_asunder.

Reference: http://www.stackoverflow.com/questions/4579908.

Note: If this method is used on Windows paths, it's recommended to os.path.splitdrive() before running this method on some path, to handle edge cases where driver letters have different meanings (example: c:path versus c:path)

Parameters **path** (*str*) – The path to split up into parts.

Returns The split path.

Return type list[str]

ways.core.testsuite module

Discover and run the Python test files in this package.

`ways.core.testsuite.`**`discover_and_run`**`()`

Look in the tests/test_*.py folder for unittests and run them all.

Module contents

High frequency modules, classes and functions to include in projects.

CHAPTER 5

Developers

5.1 Contributing

Contributions are welcome, and they are greatly appreciated! Every little bit helps, and credit will always be given.

5.1.1 Installation

To get Ways to run locally, clone the repo from online.

```
git clone http://www.github.com/ColinKennedy/ways.git
cd ways
git submodule update --init --recursive
```

Test that the repository cloned successfully by running

```
tox
```

The latest commit in the "master" branch should have passing tests.

You can also verify that your installation works by running the Ways demo file.

```
python -m ways.demo
```

Output:

```
Hello, World!
Found object, "Context"
A Context was found, congrats, Ways was installed correctly!
```

Once you see those 3 lines, you're all set to begin.

5.1.2 Reporting Issues

Before reporting issues, check to make sure that you've installed Ways and try to run its unittests. If every unittest passes and you still have your issue, please use this URL to submit your issue.

Documentation Improvements

Ways could always use more documentation, whether as part of the official Ways docs, in docstrings, or even on the web in blog posts, articles, and such.

Feature Requests And Feedback

The best way to send feedback is to file an issue at https://github.com/ColinKennedy/ways/issues.

If you are proposing a new feature:

- Explain in detail how it would work.
- Keep the scope as narrow as possible, to make it easier to implement.
- Remember that this is a volunteer-driven project, and that code contributions are welcome :)

5.1.3 Before You Submit The Issue

Check the docs before reporting an issue. It may have already been addressed.

Make sure you're running the latest version of Ways. The issue may be fixed already.

Search the issue tracker for similar issues. If you think your issue is still important enough to raise, do so, but link to the related tickets, too.

5.1.4 When You Write The Issue

1. If your problem is involved with an environment set up, please include a compressed archive (.zip/.rar/.tar/.etc) containing all of the files needed and write steps to reproduce your problem.
2. Add the output of `ways.api.trace_all_descriptor_results_info()` and `ways.api.trace_all_plugin_results_info()` as a text file or link.
3. Write a test case for your issue. It helps a lot to just pick up a test and make that test pass so that the issue won't happen again in the future.
4. Include your WAYS_PLATFORMS and WAYS_PLATFORM environment variables, if those environment variables have any information, as well as your OS and OS version.

5.1.5 Maintainer Notes

If you're considering adding features to Ways, the very first thing to do would be to clone the main repository. See *How To Install* for details.

It's recommended to read all of the documentation from start to end before making changes. But at the very least, read *API Summary*, *Getting Started* and *API Details*.

Repository Structure

Ways uses a cookiecutter tox environment. For more details, check out the GitHub repo that Ways was built from for details:

https://github.com/ionelmc/cookiecutter-pylibrary

Pull Requests

If you need some code review or feedback while you're developing the code just make the pull request.

For merging, keep these things in mind:

1. Write easy to read/maintain code.
   - K.I.S.S. Ways gets by using very few classes and very simple ideas. If you're adding a class or a complex system, think about why you think you need it, first.
   - Ways has many working parts. It tries its best to not make any assumptions about Context mapping strings or anything else. Any OS-dependent changes (like adding functions to convert "/" or "\", just as an example) will be met with caution.
2. Write tests for your changes

   At the time of writing, its coverage is over 90% so lets keep it up!
3. Explain why your pull request is needed

   This project was written by a single person, with a very specific pipeline in mind. There's bound to be ideas here that aren't going to translate as well for your pipeline needs. If you can explain what your change does and how it adds value, more power to you!

To make sure your changes work with the rest of the Ways environment, run

```
tox
```

The tox environment that Ways comes with has some commands for pylint, pydocstyle and the like. If you want to only run those, use

```
tox -e check
```

If tox passes[1], you're almost ready.

1. Update documentation when there's new API, functionality etc.
2. Add a note to `CHANGELOG.rst` about the changes.
3. Add yourself to `AUTHORS.rst`.

api.py

If the pull request contains new functions or classes, consider adding them to api.py and explain why you think they'd be a good addition.

[1] If you don't have all the necessary python versions available locally you can rely on Travis - it will run the tests for each change you add in the pull request.
It will be slower than running locally though . . .

5.1.6 Tips

To run a subset of tests:

```
tox -e envname -- py.test -k test_myfeature
```

To run all the test environments in *parallel* (you need to `pip install detox`):

```
detox
```

Python Module Index

Index