turberfield-dialogue Documentation

Release 0.47.0

D Haynes

Jun 13, 2022

CONTENTS

CHAPTER

ONE

OVERVIEW

1.1 So, you're a writer?

Turberfield-dialogue is a framework which supports screenwriting. It adopts terminology from the screenplay tradition, to define a script format which feels familiar to writing practitioners.

Turberfield comes with a rehearsal tool which lets you run through your dialogue and fine-tune your tone and pace.

We'll go over the syntax of Turberfield scene files in a moment. It is based on a system called reStructuredText. That's the same system which generated the pages you are reading now.

As well as producing dialogue, you may be called upon to define some game logic too. The Turberfield framework encourages you to write game logic in Python. Luckily, it's probably the friendliest programming language you'll find.

1.2 So, you're a developer.

If you have written no code at all yet, Turberfield by itself can provide all you need for an early prototype.

Or, you can use it as a library to provide a dialogue system for your existing Python game.

Turberfield is also your publisher. When it's time to collaborate with others or show your dialogue to an audience, you'll use the Python packaging system to distribute and install your work.

By the end of the creative process, you will need some familiarity with packaging techniques. That's not usually a subject for beginners, so I wrote this easy tutorial. If you revisit this topic from time to time, you should have learned what you need by the time your dialogue is ready to share.

CHAPTER

TWO

INSTALLATION

2.1 Choose a Text Editor

A text editor is a program for writing text and storing it in a plain format which all computers understand. This is in contrast to Microsoft Word or LibreOffice which have their own particular file formats.

If you are a developer, you may already know which you prefer. Otherwise, you should download VSCode. It's free of charge for all platforms.

2.2 Pick your Python

The version of Python you're going to use depends on your Operating System. Two examples are documented below:

- An environment on *Linux or MacOSX* with Python 3.5 installed from a package repository.
- An environment on *Microsoft Windows 8.1* with Python 3.6.1 downloaded from the Python website.

You should run Turberfield in the most recent Python available for your OS. If you find a more recent version than is shown here, then do pick that.

2.3 Create a Python virtual environment

2.3.1 Linux or MacOSX

1. Install *python3.5* using the package manager.
2. Create a new Python virtual environment:

```
$ python3.5 -m venv ~/py3.5
```

3. Upgrade your version of *pip*:

```
$ ~/py3.5/bin/pip install --upgrade pip
```

2.3.2 Microsoft Windows 8.1

1. Ensure the environment variable '*%USERPROFILE%*' points to your user directory.
2. Download and install Python 3.6.1 for Windows.
3. Create a new Python virtual environment:

```
> C:\Program Files (x86)\Python 3.6\python.exe -m venv %USERPROFILE%\py3.6
```

4. Upgrade your version of *pip*:

```
> %USERPROFILE%\py3.6\Scripts\pip install --upgrade pip
```

What's pip?

When you first install Python, it comes with only a small number of programs to run.

Pip installs packages. That is, programs and libraries. Turberfield is one such library. There are many thousands more. A community of developers puts them on the internet for us to use.

When you invoke *pip* like this, it goes out to the internet to find the package you want. It downloads it and installs it to your Python environment.

You can also tell *pip* how much of a package to install. If that package does many things, you can limit it to one particular job, or install everything it is capable of doing.

In a moment, we will install *turberfield-dialogue*, including all its *play* capabilities.

2.4 Install Turberfield Dialogue into the Python environment

On Linux and MacOSX:

```
$ ~/py3.5/bin/pip install turberfield-dialogue[play]
```

On Windows 8.1:

```
> %USERPROFILE%\py3.6\Scripts\pip install turberfield-dialogue[play]
```

2.5 Download the examples

1. Download the example source as a zipfile.
2. Unzip the archive in your home directory.
3. `cd turberfield-dialogue-master`

CHAPTER

THREE

EXAMPLE 1: BATTLE ROYAL

Turberfield comes with a couple of examples. We will start with the simplest. What we learn here will set us up for the more advanced example later.

This is our first encounter with the Turberfield rehearsal tool. We will use it to preview the action in the first example.

What are my options?

It may be that this is the first time you have launched a Python program from your computer's command line. If so, there's a couple of things to understand first.

When you launch a command line program, you do so by typing its name. You then follow that with *options* which are extra instructions to control the way the program behaves.

The Turberfield rehearsal tool takes several options. Typing them all out every time is an annoyance. So the essential ones are stored in a text file called *rehearse.cli*. You can pass this file to the program instead and it will take the options from there. You can add more from the command line at the same time by typing them afterwards in the usual way.

To instruct the rehearsal tool to load stored options, precede the path to the options file with a @ symbol.

3.1 Rehearsal

On Linux or MacOSX:

```
$ cd sequences/battle
$ ~/py3.5/bin/turberfield-rehearse @rehearse.cli
```

On Windows 8.1:

```
> cd sequences\battle
> start %USERPROFILE%\py3.6\Scripts\turberfield-rehearse @rehearse.cli
```

You can do this.

From now on, I'll assume you know how to operate the command line on your computer. Further instructions will give the Linux form of commands only, and omit the prompt character.

Here's what you should see in your terminal window. The dialogue is delivered incrementally. There's also a sound effect at the appropriate point:

```
Scratchy
        I hate the way you use me,  Itchy  !

Ol' Rusty Chopper
        **Whack!**

Itchy
        Uuurrggh!

  Itchy.state = 0
```

3.2 Script file

Let's take a peek at the file which generates the dialogue. You can open *sequences/battle/combat.rst* to see it in full. Here's the gist of it below.

```
.. entity:: FIGHTER_1
   :states: 1
   :roles: WEAPON

.. entity:: FIGHTER_2
   :types: logic.Animal
   :states: 1

.. entity:: WEAPON
   :types: logic.Tool


[FIGHTER_1]_

    I hate the way you use me, |fighter2| !

.. fx:: logic slapwhack.wav
   :offset: 0
   :duration: 3000
   :loop: 1

[WEAPON]_

    **Whack!**

[FIGHTER_2]_

    Uuurrggh!

.. property:: FIGHTER_2.state 0

.. |fighter2| property:: FIGHTER_2.name.firstname
```

If you look at the yellow highlighted sections, you'll see immediately how they correspond to lines of dialogue. Notice how they aren't allocated to characters by name. Instead, the dialogue is written for generic *roles*. Part of Turberfield's

job is to match characters to those roles.

The script file also contains other sections which do not correspond to dialogue. They are called *directives*. I will explain those in the next section.

If names be not correct…

From now on, I'm going to start being precise in what I call things. I will avoid the words *Actor* and *Character*, since they suggest a human being.

In screenplay any thing, whether animate or inanimate, can have a voice. So Turberfield calls them **Entities**.

Entities can have **attributes**. An entity with a *name* attribute is called a **Persona**. An entity with *state* attributes is called **Stateful**. In addition to those, you can define your own **types** for your entities. So long as their types match, one entity can play the **role** of another entity.

3.3 References

Alongside the script file, there is a Python (.py) file. Python files are called *modules*. They supply the entities referred to in the script. You should take a look in detail at *sequences/battle/logic.py*. Here below are its main features.

```
from turberfield.dialogue.model import SceneScript
from turberfield.dialogue.types import Persona
from turberfield.dialogue.types import Stateful


class Animal(Stateful, Persona):
    pass

class Tool(Stateful, Persona):
    pass

references = [
    Animal(name="Itchy").set_state(1),
    Animal(name="Scratchy").set_state(1),
    Tool(name="Ol' Rusty Chopper").set_state(1),
]

folder = SceneScript.Folder(
    pkg=__name__,
    description="Cartoon battle demo",
    metadata=None,
    paths=["combat.rst"],
    interludes=None
)
```

This file performs five tasks:

Lines 1 - 5
: Import what we need from Turberfield.

Lines 8 - 12
: Define some types which are necessary for the scene.

Lines 14 - 18
Create some objects to be referenced by the script. We also give them a state at the same time.

Lines 20 - 26
Declare a folder object which contains our scene script file. There are several other elements here, and we'll go into it properly later.

3.4 Type

A type is a concept from Python. You can create types with a *class* declaration in a Python module. In all these examples, we do no more than inherit behaviour from other base classes, hence the single *pass* instruction in the class body.

Notice that two of the entity declarations in the script file have a *:types:* constraint; Fighter 2 has to be some kind of Animal, and the Weapon a Tool.

3.5 State

The Battle Royal sequence makes use of *state*. Both fighters must be alive at the beginning of the scene. This is encoded as a simple integer state, which is set in the Python module when the references are created.

The entity declaration in the script file specifies the state must be 1 in order for a persona to be cast as one of the fighters in the scene.

A property directive in the scene file zeroes the state of the smitten fighter. We'll look in more detail how this works in the *Syntax guide*.

3.6 Roles

By default, Turberfield's rehearsal proceeds despite any unmatched entities. The lines will not be voiced for unmatched parts. If you want to be strict about only playing fully-cast scene files, you can specify that with an option.

In either case, if *none* of the entities in the scene can be cast, the entire scene is skipped.

An extra dimension to the casting of entities is the concept of *roles*. When roles are attached to an entity declaration it means that the persona which gets cast to play that entity becomes a candidate to play those other entities too.

In this way, a scene written as a montage of ensemble dialogue could still be delivered as a monologue were there to be only one persona available to deliver the lines.

3.7 Repeats

By default the rehearsal tool runs through the scene just once. To see the effect of roles in this example, we'll need the scene to repeat. Launch the rehearsal again, this time specifying a repetition:

```
~/py3.5/bin/turberfield-rehearse --repeat=1 @rehearse.cli
```

And you should see the carnage play out, with one inevitable winner left standing:

```
Scratchy
      I hate the way you use me,  Itchy  !

Ol' Rusty Chopper
      **Whack!**

Itchy
      Uuurrggh!

    Itchy.state = 0

Ol' Rusty Chopper
      I hate the way you use me,  Scratchy  !

Ol' Rusty Chopper
      **Whack!**

Scratchy
      Uuurrggh!

    Scratchy.state = 0
```

CHAPTER

FOUR

SYNTAX GUIDE

A Turberfield scene script file represents a sequence of dramatic action. A single file contains one or more scenes, each of which is composed of separate shots. A shot can contain dialogue, audio effects and other directives.

A Scene script adopts the reStructuredText syntax. By using a defined subset of that syntax, along with custom extensions, Turberfield defines a format which visually resembles a traditional screenplay, yet which conforms to a formal data model.

A key feature of the format is that it allows roles to be cast dynamically, such that objects must match the declared criteria for a role. It is nonetheless possible for dialogue to refer to personal attributes such as names, using a mechanism of substitution references.

4.1 Naming convention

The file name should contain lower case characters only. It should contain no whitespace or punctuation characters. Underscores should be used to represent spaces.

The file must have a suffix of **.rst**. If you'd like to distinguish between dialogue text and regular reStructuredText, then give your scene script file a **.dlg.rst** suffix.

4.2 Structure

1. The file should begin with a comment identifying it as a Turberfield scene script.
2. The file should contain metadata identifying the author(s) in a field list. The following fields are recommended:
   - author
   - date
   - copyright
3. The file must contain at least one *entity declaration* before any scene is defined.
4. The file must contain at least one **scene** section which is created by a top-level heading.
5. Each scene must contain at least one **shot** section which is created by a second-level heading.
6. Dialogue is defined by a citation in a shot section. The name given in the citation must match an entity declaration. Dialogue may contain inline markup and substitution references.
7. **Shot sections may also contain one or more of these elements:**
   - *Property directive*
   - *FX directive*

- *Condition directive*
- *Memory directive*

4.3 Elements

4.3.1 Entity declaration

`.. entity:: NAME`

The name of an entity should be in upper case.

`:roles:`
A whitespace separated sequence of other entities. A match for this entity will be considered for those roles. Optional.

`:states:`
A whitespace separated sequence of dotted paths. Each path must resolve to a Python Enum class member. In addition, a single integer value is also allowed.

The resolution rules allow for hierarchical states, eg:

- *3* matches integer states *301*, *38* and *3*.
- *Location.pub* matches *Location.pub* and *Location.pub_carpark*.

A candidate for this entity must match *all* these state criteria. Optional.

`:types:`
A whitespace separated sequence of dotted paths. Each path must resolve to a Python type. A candidate for this entity must be an instance of *at least one* of these types. Optional.

4.3.2 Property directive

`.. property:: ATTRIBUTE [VALUE]`

This directive takes no other options.

The property directive acts in two modes.

- With a single argument it is a *getter*; it returns the attribute value. This is the mode to use when defining a substitution reference.
- With two arguments it is a *setter*; it sets the attribute value. This allows you to modify entities during the delivery of dialogue.

4.3.3 FX directive

`.. fx:: PACKAGE RESOURCE`

The FX (effects) directive calls up a visual cue or a sound effect. The first argument is the dotted name of the package which contains the asset file. The second argument is the path of the file relative to the package location.

`:duration:`
Sets the duration (audio playback, display of still image). This value is in milliseconds. Optional.

`:loop:`
The number of times to play the audio or display a still image.

:offset:
Sets the point in an audio file at which playback begins. This value is in milliseconds.

:label:
A text label for the resource. May contain substitution references.

4.3.4 Condition directive

.. condition:: ATTRIBUTE VALUE

This directive takes no other options.

The condition directive specifies that a comparison be evaluated. If VALUE is supplied in parentheses, it is used as a Regular Expression. Otherwise, it is treated as a plain string, though it may contain substitution references.

The intended purpose of this directive is to mask off sections of dialogue which do not satisfy certain criteria.

4.3.5 Memory directive

.. memory:: STATE

The Memory directive saves a record to the dialogue database. STATE is the dotted path to a Python Enum class value, or else an integer.

This directive lets you capture relationships between entities and store them with a timestamp and a note of explanation.

:subject:
The name of an entity which is primarily associated with STATE. With no *object* (see below) the interpretation is that the subject is assigned the state. If object is defined, the relationship between subject, object and state is application-specific.

:object:
The name of an entity which is the object of the relationship `(subject, state, object)`. Optional.

Any paragraphs of inline content to this directive are used as a note which accompanies the record in the database. Such paragraphs may contain inline markup and substitution references.

CHAPTER

FIVE

EXAMPLE 2: CLOAK OF DARKNESS

Turberfield-dialogue is a screenplay system; it's not intended by itself to be a game framework.

Nonetheless, it strikes me that you might want to use it as a first port of call when prototyping ideas for a large game. Especially if that game is to contain a decent amount of dialogue.

So I decided to explore the degree to which this is possible. And here is my implementation of Cloak of Darkness.

The *Hello World* of adventure games.

Cloak of Darkness is a scenario which exercises several features that all game frameworks really should support.

By implementing the game, the framework author demonstrates how those features are achieved.

The game consists of three rooms. In one room is an invisible prize. But the prize is damaged by the player every time he visits the room. It turns out that the player is carrying an item which when dropped in one of the other rooms, allows the prize to be seen.

5.1 Interludes

An adventure game is interactive. The action frequently pauses to allow the player to input a command. In classic text adventures, that command is typed in to the console.

Turberfield-dialogue has a feature called *interludes*. You may have spotted that earlier; it's the fifth argument to a `Folder`. In *Example 1: Battle Royal* we put `None` which is a way of declining to use interludes.

An interlude is a function which gets called at the end of a scene script file. You can define a different function for each if you like. The function sees the folder you're using. It knows which of the scene files has just finished. It also gets to see all the references you passed in to the performance.

An interlude function's return value is the metadata for a folder object. Returning the current folder's metadata is like saying *continue*. Or you can return different metadata matching another folder and thereby branch the story.

5.2 Design

My first instinct was to create a scene file for each of the three rooms. Then all the action for a room goes in to the same file. We will repeat the sequence of three scenes over and over, with an interlude between them to take user input.

Both the player and the cloak can move location. So we will need a state to represent that. Also, the message must change every time it is damaged. That could require another state, but it turns out that a simple Python attribute will suffice.

I want the game to run in *Rehearsal*. Here's where the main problem arises. The default behaviour is to run sequentially through scenes, and deliver any dialogue possible. In this case, we only want the action to take place in the location of the player. Otherwise, we will get ghostly voices leaking in from other rooms. So we need to ensure that all other roles remain uncast outside of the current location.

In the previous example we used an integer state variable to mark the fighters as alive or dead. I'll reuse that concept. By default, every entity is masked out. As the player moves around, certain objects become active, and others go quiet.

5.3 Implementation

5.3.1 Logic

We're going through the code now. If Python is new to you, don't worry. My intention is just to introduce some essential concepts which you can work to understand later on.

The top of a Python module is where imports go:

```
import enum
from itertools import repeat
import random

from turberfield.dialogue.model import SceneScript
from turberfield.dialogue.types import DataObject
from turberfield.dialogue.types import Stateful
```

Next we declare an enumeration state which will define the location of the player and the cloak:

```
@enum.unique
class Location(enum.Enum):
    foyer = 0
    bar = 1
    cloakroom_floor = 2
    cloakroom_space = 3
    cloakroom_hook = 4
```

There are no Personas in this game; none of the voices has a name. But they do have state, and one of them needs attributes. The useful types to inherit from will be *Stateful* and *DataObject*.

Each of the entities in the game gets its own class declaration:

```
class Narrator(Stateful):
    pass

class Cloak(Stateful):
    pass
```

(continues on next page)

(continued from previous page)

```
class Prize(Stateful, DataObject):
    pass
```

So now we can declare an ensemble of entities, setting attributes and initial state where appropriate:

```
ensemble = [
    Narrator().set_state(Location.foyer),
    Cloak().set_state(Location.foyer).set_state(1),
    Prize(message="You win!")
]
```

We will be taking user input and trying to interpret commands. Here is the world's dumbest text parser. It returns the first letter of the last word typed into the console:

```
def parse_command(cmd):
    try:
        return cmd.strip().split(" ")[-1][0].lower()
    except:
        return None
```

We want user input at the end of every turn. That's done in a single interlude function. Should the game grow any larger, it would be better to give each file its own custom function, but this is good enough for an example. I'm just going to throw the code at you and see how you get on:

```
def interaction(folder, index, ensemble, *args, cmd="", log=None, **kwargs):
    narrator, cloak, prize, *others = ensemble
    locn = narrator.get_state(Location)
    action = None
    if locn == Location.foyer:
        while action not in ("s", "w", "q"):
            action = parse_command(cmd or input("Enter a command: "))
        if action == "s":
            narrator.set_state(Location.bar)
            if cloak.get_state(Location) == locn:
                prize.set_state(0)
            else:
                prize.set_state(1)
        elif action == "w":
            narrator.set_state(Location.cloakroom_space)
            cloak.set_state(1)
        else:
            return None
    elif locn == Location.bar:
        while action != "n":
            action = parse_command(cmd or input("Enter a command: "))

        narrator.set_state(Location.foyer)
        prize.message = prize.message.replace(
            random.choice(prize.message), " ", 1
        )
        prize.set_state(0)
    elif locn == Location.cloakroom_space:
```

(continues on next page)

(continued from previous page)

```
        while action not in ("c", "h", "e"):
            action = parse_command(cmd or input("Enter a command: "))
        if action == "c":
            if cloak.get_state(Location) == Location.cloakroom_space:
                cloak.set_state(Location.cloakroom_floor)
            else:
                cloak.set_state(Location.cloakroom_space)
        elif action == "h":
            cloak.set_state(Location.cloakroom_hook)
        else:
            narrator.set_state(Location.foyer)
            if cloak.get_state(Location) != locn:
                cloak.set_state(0)

    if cloak.get_state(Location) == locn:
        cloak.set_state(narrator.get_state(Location))
        cloak.set_state(1)

    return folder.metadata
```

So now we can declare the objects *turberfield-rehearse* needs to see; a collection of all our Python references and a folder object with details of the game:

```
references = ensemble + [Location]

folder = SceneScript.Folder(
    pkg=__name__,
    description="The 'Hello World' of text games.",
    metadata={},
    paths=["foyer.rst", "bar.rst", "cloakroom.rst"],
    interludes=repeat(interaction)
)
```

Coding.

Python is a pretty easy language to read, and so far I've been relying on that to communicate the essence of how all this works. We have reached a point now that you may need to take time over certain aspects of the code to fully understand what is going on.

I recommend you explore the Python manual. First, get to know its structure; how it separates the fundamentals of the language from details of specific modules which you discover when you realise you need them.

To begin with, check out the random module which is very straightforward. After that, use the module index to find the documentation for *Enum*.

5.3.2 Dialogue

Here's where I stop explaining each component of the game. When it comes to understanding the dialogue, it's best just to study the *.rst* files in *sequences/cloak*. As a taster, here's what the dialogue for the first room looks like. It's probably the simplest of the three.

```
.. entity:: NARRATOR
   :types: logic.Narrator
   :states: logic.Location.foyer

.. entity:: CLOAK
   :types: logic.Cloak
   :states: logic.Location.foyer

After the fire, a Magician returns
~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~

From where you stand
--------------------

[NARRATOR]_

    This place no longer looks much like a hotel. This would have been the foyer, though.
    You can see the footprint of a grand reception desk running down one side
    of the floor.

[NARRATOR]_

    The room has been stripped of all it once contained.

Checking your person
--------------------

[CLOAK]_

    You are wearing a long cloak, which gathers around you. It feels furry,
    like velvet, although that's hard to tell by looking. It is so black
    that its folds and textures cannot be perceived.

[CLOAK]_

    It seems to swallow all light.

.. memory:: logic.Location.foyer
   :subject: NARRATOR

   The Player visited the foyer.

Looking around
--------------

[NARRATOR]_
```

(continues on next page)

(continued from previous page)

```
    To the North, the door by which you first entered is stuck fast.

[NARRATOR]_

    There are other doors to the South and West.
```

5.4 Action

You can run the game in a similar manner to the previous example:

```
cd sequences/cloak
~/py3.5/bin/turberfield-rehearse @rehearse.cli
```

5.5 Memory

We saw for the first time above the use of a *Memory directive*. The game scatters them throughout the action. The result is that a record of the player's progress builds up in the dialogue database.

The database Turberfield uses for this is SQLite3. You can access this database via Python's own SQLite3 module. Or you can install a command line tool and issue queries that way. Try this to get a report of the passage of a game session:

```
sqlite3 cloak.sl3

sqlite> select s.name, state.name, note.text
   ...> from state join touch on state.id = touch.state
   ...> join entity as s on touch.sbjct = s.id
   ...> left outer join entity as o on touch.objct = o.id
   ...> left outer join note on note.touch = touch.id;

Narrator|foyer          |The Player visited the foyer.
Cloak   |bar            |The Player wore the cloak in the bar.
Narrator|foyer          |The Player visited the foyer.
Cloak   |cloakroom_floor|The Player dropped the cloak.
Narrator|foyer          |The Player visited the foyer.
Cloak   |bar            |The Player wore the cloak in the bar.
Narrator|foyer          |The Player visited the foyer.
Cloak   |cloakroom_hook |The Player hung the cloak on a hook.
Narrator|foyer          |The Player visited the foyer.
Prize   |bar            |The Player read the message as " Yo  w n! ".
Narrator|foyer          |The Player visited the foyer.
```

CHAPTER

SIX

CLI REFERENCE

6.1 turberfield-dialogue

A utility to generate a printable screenplay.

The output is HTML. It is styled to be an input to WeasyPrint, so you can ultimately create a script in PDF format.

There are command line options to change the timing of dialogue, to repeat the action, and to control the number of roles an entity may take.

Example (see the episode Blue Monday):

```
turberfield-dialogue    --references=bluemonday78.logic:references
                        --folder=bluemonday78.logic:ray
                        --folder=bluemonday78.logic:justin
                        --folder=bluemonday78.logic:local
                        --roles=1
                        --repeat=0
                        --strict
```

```
usage: turberfield-dialogue [-h] [--version] [-v] [--log LOG_PATH]
                            [--references REFERENCES] [--folder FOLDER]
                            [--roles ROLES] [--strict] [--repeat REPEAT]
                            [--pause PAUSE] [--dwell DWELL]
```

6.1.1 options

--version	Print the current version number
-v, --verbose	Increase the verbosity of output
--log	Set a file path for log output
--references	Give an import path to a list of Python references.
--folder	Give a sequence of import paths to SceneScript folders.
--roles	The number of roles [1] permitted for each member of cast.
--strict	Only perform fully-cast scene files.
--repeat	Repeat the performance [0] times.
--pause	Time in seconds [1.2] to pause after a line.
--dwell	Time in seconds [0.3] to dwell on each word.

6.2 turberfield-rehearse

Caution: This tool has a *web mode* which is experimental. It may not work perfectly in your web browser.

It also presents a potential security risk while it is running, since its CGI interface facilitates code execution on your computer.

Always check that your PC firewall does not permit outside access to the port configured by the program options. If in doubt, disconnect your PC from all networks while web mode is in operation.

CHAPTER

SEVEN

API REFERENCE

This API lets you customise the rehearsal of a `Folder` of Turberfield dialogue. You can modify how this performance is presented in your game by customising or replacing a *handler* function which processes *events* from the scene script files.

If you need to integrate with a game event loop, then create a `Performer` object instead. This separates event generation from sequencing.

The *scene scripts* themselves also have an API, allowing you to interact with the process by which entities are selected and cast to roles.

There is also a simple `Matcher` class which you can use to compare folder metadata.

7.1 Folders

SceneScript.**Folder (pkg, description, metadata, paths, interludes)**

Parameters

- **pkg** – The dotted (importable) name of the package which installs the scene script folder.

  Absent proper packaging, you must set this parameter to *__name__*. The Python module which declares the folder will act as the anchor location for scene files, and all referencing paths will need to be made relative to the module.

- **description** (*str*) – A free text description of the contents of the folder.

- **metadata** – An optional sequence or mapping containing application-specific metadata.

  This parameter is for the purposes of searching or filtering collections of folders against particular criteria.

- **paths** – A list of strings, each of which is the path to a scene script file relative to the object declared in the parameter **pkg**.

  Path separator is always "/" notwithstanding the local Operating System.

- **interludes** – A sequence of function objects. The sequence should be such as to provide one object for each of the scene script files declared in the parameter **paths**.

  A function object will be called when its corresponding scene script file has been performed.

  See *Interludes* for the required signature of an interlude function object.

7.2 Scene scripts

class turberfield.dialogue.model.**SceneScript**(*fP*, *metadata=None*, *doc=None*)

Gives access to a Turberfield scene script (.rst) file.

This class allows discovery and classification of scene files prior to loading them in memory.

Once loaded, it allows entity selection based on the role definitions in the file. Casting a selection permits the script to be iterated as a sequence of dialogue items.

classmethod scripts(*pkg*, *metadata*, *paths=[]*, ***kwargs*)

This class method is the preferred way to create SceneScript objects.

Parameters

- **pkg** (*str*) – The dotted name of the package containing the scripts.
- **metadata** – A mapping or data object. This parameter permits searching among scripts against particular criteria. Its use is application specific.
- **paths** (*list(str)*) – A sequence of file paths to the scripts relative to the package.

You can satisfy all parameter requirements by passing in a *Folder* object like this:

```
SceneScript.scripts(**folder._asdict())
```

The method generates a sequence of *SceneScript* objects.

static read(*text*, *name=None*)

Read a block of text as a docutils document.

Parameters

- **text** (*str*) – Scene script text.
- **name** (*str*) – An optional name for the document.

Returns
A document object.

select(*personae*, *relative=False*, *roles=1*)

Select a persona for each entity declared in the scene.

Parameters

- **personae** – A sequence of Personae.
- **relative** (*bool*) – Affects imports from namespace packages. Used for testing only.
- **roles** (*int*) – The maximum number of roles allocated to each persona.

Returns
An OrderedDict of {Entity: Persona}.

cast(*mapping*)

Allocate the scene script a cast of personae for each of its entities.

Parameters
mapping – A dictionary of {Entity, Persona}

Returns
The SceneScript object.

run()

Parse the script file.

Return type
Model

7.3 Events

class turberfield.dialogue.model.**Model**(*fP*, *document*)

This class registers the necessary extensions to the docutils document model.

It also defines the types which are returned on iterating over a scene script file.

Model.**Shot (name, scene, items)**

An event which signals the beginning of a shot in a scene.

Model.**Property (entity, object, attr, val)**

An event which signals a property is to be accessed.

Model.**Audio (package, resource, offset, duration, loop)**

An event which signals an audio cue.

Model.**Still (package, resource, offset, duration, loop, label, width, height)**

An event which signals a still image cue.

Model.**Video (package, resource, offset, duration, loop, label, width, height, poster, url)**

An event which signals a video cue.

Model.**Memory (subject, object, state, text, html)**

An event which signals a memory directive.

Model.**Line (persona, text, html)**

An event which signals a line of dialogue.

Model.**Condition (object, attr, val, operator)**

An event which evaluates a conditional expression.

7.4 Interludes

An interlude is a callable object (either a function, an instance method or a Python object with a callable interface).

It is called by a handler at the end of the performance of a scene script file. That is the *current scene* file as referred to below.

Here is an example to show the signature of parameters required.

def my_interlude(folder, index, ensemble, log=None, loop=None):

Parameters

- **folder** – A *Folder* object.
- **index** (*int*) – The index position into **folder.paths** of the current scene script file.

- **ensemble** – A sequence of Python objects. It is guaranteed to contain all the objects cast to roles in the current scene. It will be used to select entities for the next.
- **log** – If supplied, this will be a `logging.Logger` object which should be used in preference over any other for logging messages from within your interlude function.
- **loop** – If supplied, this will be an instance of `asyncio.BaseEventLoop`. That will signal to your function that it operates in an asynchronous environment and that no blocking function should be called within it.

Returntype
dict

Returning this object gives you the option to control branching of your narrative.

The dictionary data you pass back is matched against the metadata of each folder. The folder whose metadata matches closest is the next folder to run.

Return **None** to stop the performance.

7.5 Handler

class turberfield.dialogue.handlers.**TerminalHandler**(*terminal*, *dbPath=None*, *pause=1.2*, *dwell=0.3*, *log=None*)

The default handler for events from scene script files. It generates output for a console terminal.

The class is written to be a callable, stateful object. Its *__call__* method delegates to handlers specific to each type of event. You can subclass it and override those methods to suit your own application.

Parameters

- **terminal** – A stream object.
- **dbPath** (*str*) – An optional URL to the internal database.
- **pause** (*float*) – The time in seconds to pause on a line of dialogue.
- **dwell** (*float*) – The time in seconds to dwell on a word of dialogue.
- **log** – An optional log object.

static handle_audio(*obj*, *wait=False*)

Handle an audio event.

This function plays an audio file. Currently only *.wav* format is supported.

Parameters

- **obj** – An *Audio* object.
- **wait** (*bool*) – Force a blocking wait until playback is complete.

Returns
The supplied object.

handle_interlude(*obj*, *folder*, *index*, *ensemble*, *loop=None*, ***kwargs*)

Handle an interlude event.

Interlude functions permit branching. They return a folder which the application can choose to adopt as the next supplier of dialogue.

This handler calls the interlude with the supplied arguments and returns the result.

Parameters

- **obj** – A callable object.
- **folder** – A *Folder* object.
- **index** (*int*) – Indicates which scene script in the folder is being processed.
- **ensemble** – A sequence of Python objects.
- **branches** – A sequence of *Folder* objects. from which to pick a branch in the action.

Returns
A *Folder* object.

handle_line(*obj*)

Handle a line event.

This function displays a line of dialogue. It generates a blocking wait for a period of time calculated from the length of the line.

Parameters
obj – A *Line* object.

Returns
The supplied object.

handle_memory(*obj*)

Handle a memory event.

This function accesses the internal database. It writes a record containing state information and an optional note.

Parameters
obj – A *Memory* object.

Returns
The supplied object.

handle_property(*obj*)

Handle a property event.

This function will set an attribute on an object if the event requires it.

Parameters
obj – A *Property* object.

Returns
The supplied object.

handle_scene(*obj*)

Handle a scene event.

This function applies a blocking wait at the start of a scene.

Parameters
obj – A *Shot* object.

Returns
The supplied object.

handle_scenescript(*obj*)

Handle a scene script event.

Parameters
obj – A *Folder* object.

Returns
The supplied object.

handle_shot(*obj*)

Handle a shot event.

Parameters
obj – A *Shot* object.

Returns
The supplied object.

7.6 Matcher

class turberfield.dialogue.matcher.**Matcher**(*folders=None*)

static mapping_key(*obj*)

A keying function which allows nested objects to be sorted.

__init__(*folders=None*)

Match Turberfield Folders by their metadata.

This class has methods to normalise arbitrary dictionaries. It provides a search API, so you can discover which folders are a metadata match.

Parameters
folders – A sequence of *turberfield.dialogue.model.SceneScript.Folder* objects.

options(*data*)

Generate folders to best match metadata.

The results will be a single, perfectly matched folder, or the two nearest neighbours of an imperfect match.

Parameters
data (*dict*) – metadata matching criteria.

This method is a generator. It yields *turberfield.dialogue.model.SceneScript.Folder* objects.

7.7 Performer

class turberfield.dialogue.performer.**Performer**(*folders*, *ensemble*)

property stopped

Is *True* when none of the folders can be cast, *False* otherwise.

__init__(*folders*, *ensemble*)

An object which can select actors for a scene and run a performance.

Parameters

- **folders** – A sequence of *Folder* objects.
- **ensemble** – A sequence of Python objects.

run(*react=True*, *strict=True*, *roles=1*)

Select a cast and perform the next scene.

Parameters

- **react** (*bool*) – If *True*, then Property directives are executed at the point they are encountered. Pass *False* to skip them so they can be enacted later on.
- **strict** (*bool*) – Only fully-cast scripts to be performed.
- **roles** (*int*) – Maximum number of roles permitted each character.

This method is a generator. It yields events from the performance.

If a *Condition* is encountered, it is evaluated. No events are generated while the most recent condition is False.

A new *Shot* resets the current condition.

7.8 Player

turberfield.dialogue.player.**rehearse**(*folders*, *references*, *handler*, *repeat=0*, *roles=1*, *strict=False*, *loop=None*)

Cast a set of objects into a sequence of scene scripts. Deliver the performance.

Parameters

- **folders** – A sequence of *turberfield.dialogue.model.SceneScript.Folder* objects.
- **references** – A sequence of Python objects.
- **handler** – A callable object. This will be invoked with every event from the performance.
- **repeat** (*int*) – Extra repetitions of each folder.
- **roles** (*int*) – Maximum number of roles permitted each character.
- **strict** (*bool*) – Only fully-cast scripts to be performed.

This function is a generator. It yields events from the performance.

CHAPTER

EIGHT

PUBLISHING

The demo examples you've seen so far were arranged as standalone directories containing a Python module and some scene script files.

You can get started quickly by working this way, but before your screenplay is ready, you need to have properly configured it as a Python package.

Packaging gives you the following advantages:

- *Versioning*
- *Attribution*
- *Distribution*
- *Installability*
- *Dependency management*
- *Discoverability*

8.1 Checklist

Turning your screenplay into a package might be a pain the first time you do it. But you'll reap the benefits after that. Here's what you have to do.

1. *Organise your project directory*
2. *Make a manifest*
3. *Write a README file*
4. *Write the setup.py*

8.1.1 Organise your project directory

Suppose your screenplay, **mydrama** is in a single directory of that name. You have three scene script files; *begin.rst*, *middle.rst*, and *end.rst*. You have an idea for a soundtrack you call *theme.wav*. And there is one Python module called *logic.py*. You have saved some options as a file called *rehearse.cli*:

```
mydrama
├── begin.rst
├── middle.rst
├── end.rst
├── theme.wav
```

(continues on next page)

(continued from previous page)

```
├── logic.py
└── rehearse.cli
```

Now create four more empty files as follows:

```
├── __init__.py
├── MANIFEST.in
├── README.rst
└── setup.py
```

There is nothing more to do to *__init__.py*. It stays empty. We will deal with the other three in turn.

Important: The naming conventions for Python packages are quite strict. Your directory name should use only lower case letters. If you want to signify a space in the directory name, use an underscore.

Also, **never use the word 'turberfield' in your package name**. It's for software tooling only.

8.1.2 Make a manifest

The *MANIFEST.in* file decides which of your source files get installed. It can filter out any project files created by your text editor, cache files and the like. It should look like this:

```
recursive-include . *.cli
recursive-include . *.rst
recursive-include . *.wav
```

8.1.3 Write a README file

The *README.rst* file is an opportunity to describe your drama to potential collaborators. It is a reStructuredText file, so you can include hyperlinks and other useful structures.

At a minimum, this file should contain your name, email address and an assertion of your copyright. Other details are up to you.

8.1.4 Write the setup.py

setup.py is like an electronic form which tells the packaging system everything about your project. Here is the standard boilerplate you should use.

```
#!/usr/bin/env python
# encoding: UTF-8

from setuptools import setup
import os.path

__doc__ = open(
    os.path.join(os.path.dirname(__file__), "README.rst"),
    "r"
).read()
```

(continues on next page)

(continued from previous page)

```
setup(
    name="mydrama",
    version="0.1.0",
    description="A dramatic screenplay",
    author="Ernest Scribbler",
    author_email="escribbler@zmail.com",
    url="http://pypi.python.org/pypi/mydrama",
    long_description=__doc__,
    classifiers=[
        "Framework :: Turberfield",
        "Operating System :: OS Independent",
        "Programming Language :: Python :: 3",
        "License :: Other/Proprietary License",
    ],
    packages=["mydrama"],
    package_dir={"mydrama": "."},
    include_package_data=True,
    install_requires=["turberfield-dialogue"],
    zip_safe=True,
    entry_points={}
)
```

Of course, you'll need to alter some details to match the name of your particular project, here:

```
name="mydrama",
```

… and here:

```
packages=["mydrama"],
package_dir={"mydrama": "."},
```

In the next few sections, we'll customise a little further.

8.2 Versioning

As soon as other people begin to use your dialogue, you'll need to give them a way of deciding whether they want to use your latest rewrite or to stick with an earlier revision. Every release of your work will have a version number to identify it.

You declare the version in the *setup* parameters in *setup.py*:

```
version="0.1.0",
```

The three digits reflect the significance of any new change:

- Trivial fixes increment the rightmost digit.
- Significant changes increment the middle version field. This is the most frequent case; the number can go as high as you like, even into the hundreds.
- Major changes which are incompatible with previous versions require an increment to the leftmost digit.

8.3 Attribution

I'm guessing your name is not Ernest Scribbler. If it is, write in and let me know! Otherwise, you'll change the following parameters to match your online identity:

```
author="Ernest Scribbler",
author_email="escribbler@zmail.com",
```

8.4 Distribution

The command to create a *distribution* of your project is this:

```
~py3.5/bin/python setup.py sdist
```

The packaging system creates an installable for you. You'll find it at *dist/mydrama-0.1.0.tar.gz* or *dist/mydrama-0.1.0.zip*, depending on your OS.

You can upload that file to a package repository. The most popular is PyPI but there are alternatives, such as Gemfury.

So you'll need to declare the correct URL to your package once it gets up there:

```
url="http://pypi.python.org/pypi/mydrama",
```

This is a bit of a chicken-and-egg situation of course. You'll have to anticipate what the URL is going to be before you upload it, or else you'll have a misprint in the first release which you'll need to fix afterwards.

8.5 Installability

With your work properly packaged, you can be confident that others can start using it with a minimum of fuss.

If you upload it to PyPI, *pip* will go out and fetch it:

```
~/py3.5/bin/pip install mydrama
```

Or you could send your package file by email or on a USB stick. Then the install command targets the package file like this:

```
~/py3.5/bin/pip install mydrama-0.1.0.tar.gz
```

8.6 Dependency management

Your package gets to declare which other Python libraries it needs to run. I already gave you the one essential dependency:

```
install_requires=["turberfield-dialogue"],
```

It's quite possible that your *logic.py* might rely on some other library to do a particular job. Perhaps you've written a role for a banker who needs to calculate loan interest.

Whatever PyPI package you add to this list will be automatically installed with your screenplay and available for use from your Python modules.

8.7 Discoverability

Publishing your work is a crucial step. But as well as that, you have to advertise. When a game developer puts out the call for some dramatic dialogue, you want to be able to say, 'Yes, there's a scene for that. I wrote it. Here it is.'

So now you need to create a unique global id for the scene you just wrote.

Python helps you here. It has a standard module called *uuid*, which is short for *unique user id*. Here's how you use it to generate a one-time code to identify a folder of scenes you just created:

```
~/py3.5/bin/python -c"import uuid; print(uuid.uuid4().hex)"
```

What you get back is a 32-character code which looks a bit like this:

```
c.1de5c.3f5a4abe..937.7.6e55a.8e
```

I put dots in it so you wouldn't cheat and copy mine. Dots are illegal. Make your own!

Now you go back to *setup.py* and edit the *entry_points* parameter. Like this:

```
entry_points={
    "turberfield.interfaces.folder": [
        "c.1de5c.3f5a4abe..937.7.6e55a.8e = mydrama.logic:folder",
    ],
},
```

Doing this advertises your folder so it can be discovered and used during the course of a game.

CHAPTER

NINE

PERFORMING

9.1 Making a name for yourself

Congratulations on self-publishing your screenplay. You can build on that and start to socialise the use of the name you chose for your project.

Remember way back when you were putting *__name__* as the **pkg** argument to declare your *Folder* object? No need to do that any more. *mydrama* (or whatever you picked instead) is the name of the package now.

Likewise in scene script files, if there's a particular type you specify for an entity, that will be *mydrama.logic.VeterinarySurgeon* and so on. And because you have published your work, the whole world knows what you mean by that.

9.2 Getting discovered

Here's how a Python developer, after installing your package, might look for some dialogue suited to his modern reimagining of some Shakespearian tragedy:

```
from turberfield.utils.misc import gather_installed

guid, folder = next(
    (i for i in gather_installed("turberfield.interfaces.folder")
    if "betrayal" in v.metadata),
    (None, None)
)
```

9.3 Constraining entity selection

One last tip. The *rehearse()* function has been good to us. But it is very forgiving in the way it allows even minimally-cast scenes to play through. Sometimes we want all or nothing. Here is a way to pre-filter scenes so that only those fully cast are performed.

```
def is_fully_cast(folder, references):
    for script in SceneScript.scripts(**folder._asdict())
        with script as dialogue:
            selection = dialogue.select(references)
            if all(selection.values()):
                continue:
```

(continues on next page)

(continued from previous page)

```
            else:
                return False
    return True
```

CHAPTER

TEN

CHANGE LOG

10.1 0.47.0

- Allow raw html directive outside the context of a shot.

10.2 0.46.1

- Fix handling of missing substitution definitions.

10.3 0.45.0

- Basic implementation of reStructuredText style substitution definitions.

10.4 0.44.0

- Special handling of footnotes to allow formatting for print.

10.5 0.43.0

- Better handling of missing substitution references.

10.6 0.42.0

- Better handling of duplicate scene names.

10.7 0.41.1

These changes are intended to allow dialogue file processing to continue despite missing object references. This is to accomodate a writing workflow which begins with regular *.rst* files, and progressively develops them to *.dlg.rst* scene scripts.

- Switch from standard library logging to using the LogManager and LogAdapter classes from turberfield-utils.
- Add a custom LogAdapter for colourized output.
- Add path and line data to model objects.
- Improve tolerance of references lacking persona.
- Improve tolerance of missing citations.
- Improve reporting of line numbers where there is an error in property substitution.
- Minimise package dependencies (blessings and simpleaudio are optional now).

10.8 0.40.0

- Fix a bug preventing duplicate shot names.

10.9 0.39.0

- *Model.Still* gets width and height.
- Implement *Model.Video*.

10.10 0.38.0

- Fix implementation of previous feature.

10.11 0.37.0

- Allow object assignment via property setter.

10.12 0.36.0

- Fix a bug with regex matching of integer state.

10.13 0.35.0

- Better collaborative multiple inheritance.

10.14 0.34.0

- Model implements format string style substitution of references.

10.15 0.33.0

- Use internal docutils type for SceneScript settings.

10.16 0.32.0

- Fix an issue on updating to docutils 0.17.

10.17 0.31.0

- Fix unit tests for Condition when not skipped.
- Permit regular expressions in Condition values.
- *Performer.allows* implements regular expression match where necessary.

10.18 0.30.0

- Dialogue model permits multiple dots in a condition specifier.
- Performer implements format string style evaluation of conditions.

10.19 0.29.0

- Improved handling of AttributeError when substituting a persona reference.

10.20 0.28.0

- Fixed a bug where the current speaker would carry over from a previous block quote.

10.21 0.27.0

This release comprises a refactoring of the parser model. You now get more flexibility, but you should check your existing projects to see if they are any changes in rendering.

- The *raw:: html* directive is now supported.
- Bullet lists are now recognised and rendered as HTML unordered lists.
- The requirement for two sections has been relaxed, allowing you to render document fragments.

10.22 0.26.0

- When building the HTML for a dialogue Line, characters are now correctly escaped as HTML5 entities.

10.23 0.25.0

- Hyperlinks are now properly rendered as Line HTML.
- The following rST/HTML5 equivalences are implemented:
  - Emphasis directives (* markup) rendered as HTML with <*em*> tags.
  - Strong directives (** markup) rendered as HTML with <*strong*> tags.
  - Literal directives (`` markup) rendered as HTML with <*pre*> tags.

10.24 0.24.0

- *Performer.react* now sets state on the subject of a memory when there's no defined object.
- Fix the interlude in *Cloak* so it works properly in rehearsal.

10.25 0.23.0

- *Stateful.set_state* now takes multiple positional arguments.

10.26 0.22.0

- Fix a bug in creating a Persona from an Assembly

10.27 0.21.0

- The *fx* declaration now has a *label* option. A label may may contain substitution references.
- *Still* cues now have a label attribute which takes its value from the *fx* declaration. The main use case for this is to provide content for the *alt* attribute of an HTML *img* tag.

10.28 0.20.0

- *SceneScript.Folder* interludes may be *None*.

10.29 0.19.0

- The *fx* declaration now generates Audio and Stills
- Added documentation for the *Matcher* class.
- Added guidance on alternative for file suffix.

10.30 0.18.0

- Added a Matcher class which can select folders by their metadata.
- *rehearse* function uses the matcher to branch to different folders.
- *turberfield-dialogue* utility uses the matcher likewise.
- **Interludes from now on must return a metadata dictionary**. Fixed the documentation and demo scenarios accordingly.
- Fixed a bug affecting the TerminalHandler when *simpleaudio* is not available.
- Simplified the documentation relating to VSCode.

10.31 0.17.0

- Fixed a bug in *Performer* which affected *condition* directives.

10.32 0.16.0

- *Performer* allows *condition* directives to access object *state*.

10.33 0.15.0

- Added the *condition* directive.

10.34 0.14.0

- *turberfield-dialogue* tool calls an interlude function after every scene file.

10.35 0.13.0

- DataObject *id* attribute is now a *uuid.UUID* object.
- The second argument to a property directive may be a substitution reference
- Added a code example for narrative resource discovery.

10.36 0.12.0

- Refactored the *rehearse* function so it uses *Performer*. Its first argument is now documented as a sequence. Legacy behaviour is preserved.

10.37 0.11.0

- Field lists at the document level are available via the *metadata* attribute of the model.
- Substitution references to Python values are properly resolved in the bodies of field lists.
- There is a new utility, *turberfield-dialogue* for producing a printable screenplay.
- The viewer module now registers all references with *turberfield.utils.assembly.Assembly*.
- The *Performer* class is now part of the public API.

10.38 0.10.1

- Changelog fixes.

10.39 0.10.0

- Substitution references are now permitted in the *resource* argument to an FX directive.

10.40 0.9.0

- *Turberfield.dialogue.performer* and matching tests implement the new Performer class. This was first prototyped in the *bluemonday78* episode of Addison Arches.

10.41 0.8.0

- *turberfield-rehearse* **–web** option works tolerably in Firefox.
- Added **strict** mode for casting a rehearsal.
- Interludes now see a sequence of folders they may **branch** to.
- State matching is hierarchical; '31' matches a criterion of '3'.
- genindex

Turberfield is the family name for a bunch of software components which support game development. This particular package is *turberfield-dialogue*. It helps you create dramatic dialogue or screenplay.

You can read the full documentation online.

CHAPTER

ELEVEN

ROADMAP

Turberfield is a *S* ur *PRISE*.

Semantic
Turberfield can be understood by machines or human beings.

Persistent
Turberfield can be stopped and saved for later.

Reusable
Turberfield can be turned into something else.

Interactive
Turberfield listens to what you say.

Simulation
Turberfield knows what it's talking about.

for Economics
…or Education, or Entertainment. Turberfield is deadly serious. And only a game.

What's missing? *UR*!

- If you've spotted a bug in Turberfield, please let me know so I can fix it.
- If you think Turberfield lacks a feature, you can help drive development by describing your Use Case.

In either event, please visit the project's issue tracker.

Author
tundish

PYTHON MODULE INDEX

t

INDEX

Symbols

A

C

E

F

H

L

M

O

P

R

S

T

V