Tale Documentation

Release 4.4

Irmen de Jong

Sep 03, 2018

Contents of this manual:


Tale

CHAPTER 1

What is Tale?

It is a library for building Interactive Fiction, mudlibs and muds in Python.

It is some sort of cross-breed between LPMud, CircleMud/DikuMud, and Infocom™ Z-machine.

Tale requires Python 3.5 or newer. (If you have an older version of Python, stick to Tale 2.8 or older, which still supports Python 2.7 as well)

You can run Tale in console mode, where it is a pure text interface running in your console window. But you can also run Tale in a simple GUI application (built with Tkinter) or in your web browser.

Note: The multi-user aspects are fairly new and still somewhat incomplete. Until recently, the focus has been on the (single player) interactive fiction things. However if my server is up, you can find running MUD instances here: http://www.razorvine.net/tale/ and here: http://www.razorvine.net/circle/

Note: This documentation is still a stub. I hope to write some real documentation soon, but in the meantime, use the source, Luke.

Tale can be found on Pypi as tale. The source is on Github: https://github.com/irmen/Tale

1.1 Getting started

Install tale, preferably using `pip install tale`. You can also download the source, and then execute `python setup.py install`.

Tale requires the appdirs library to sensibly store data files such as savegames.

It requires the smartypants library to print out nicely formatted quotes and dashes.

It requires the colorama library to print out text accents (bold, bright, underlined, reversevideo etc).

It requires the serpent library to be able to save and load game data (savegames).

(All of these libraries should be installed automatically if you use pip to install tale itself)

Optionally, you can install the prompt_toolkit library for a nicer console text interface experience, but this one is not strictly required to be able to run.

After all that, you'll need a story to run it on (tale by itself doesn't do anything, it's only a framework to build games with). There's a tiny demo embedded in the library itself, you can start that with:

```
python -m tale.demo.story
```

You can add several command line options:

- `--gui` add this to get a GUI interface
- `--web` add this to get a web browser interface
- `--mud` add this to launch the demo game as mud (multi-user) server

Fool around with your pet and try to get out of the house. There's a larger demo story included in the source distribution, in the `stories` directory. But you will have to download and extract the source distribution manually to get it.

Start the demo story using one of the supplied start scripts. You don't have to install Tale first, the script can figure it out.

You can also start it without the script and by using the tale driver directly, but then it is recommended to properly install tale first. This method of launching stories won't work from the distribution's root directory itself.

Anyway, the command to do so is:

```
$ python -m tale.main --game <path-to-the-story/demo-directory>`

# or, with the installed launcher script:
$ tale-run --game <path-to-the-story/demo-directory>`
```

You can use the `--help` argument to see some help about this command. You can use `--gui` or `--web` to start the GUI or browser version of the interface rather than the text console version. There are some other command line arguments such as `--mode` that allow you to select other things, look at the help output to learn more.

The story might prompt you with a couple of questions: Choose not to load a saved game (you will have none at first start anyway). Choose to create a default player character or build a custom one. If you choose *wizard privileges*, you gain access to a whole lot of special wizard commands that can be used to tinker with the internals of the game.

Type `help` and `help soul` to get an idea of the stuff you can type at the prompt.

You may want to go to the Town Square and say hello to the people standing there:

```
>> look

  [Town square]
  The old town square of the village.  It is not much really, and narrow
  streets quickly lead away from the small fountain in the center.
  There's an alley to the south.  A long straight lane leads north towards
  the horizon.
  You see a black gem, a blue gem, a bag, a box1 (a black box), a box2 (a
  white box), a clock, a newspaper, and a trashcan.  Laish the town crier,
  ant, blubbering idiot, and rat are here.

>> greet laish and the idiot

  You greet Laish the town crier and blubbering idiot.  Laish the town
```

(continues on next page)

(continued from previous page)

```
  crier says: "Hello there, Irmen."  Blubbering idiot drools on you.

>> recoil

  You recoil with fear.

>>
```

1.2 Features

A random list of the features of the current codebase:

- requires Python 3.5 or newer
- game engine and framework code is separated from the actual game code
- single-player Interactive Fiction mode and multi-player MUD mode
- selectable interface types: text console interface, GUI (Tkinter), or web browser interface
- MUD mode runs as a web server (no old-skool console access via telnet or ssh for now)
- can load and run games/stories directly from a zipfile or from extracted folders.
- wizard and normal player privileges, wizards gain access to a set of special 'debug' commands that are helpful while testing/debugging/administrating the game.
- the parser uses a soul based on the classic LPC-MUD's 'soul.c' from the late 90's
- the soul has 250+ 'emotes' such as 'bounce', 'shrug' and 'ponder'.
- it knows 2200+ adverbs that you can use with these emotes. It does prefix matching so you don't have to type it out in full (gives a list of suggestions if multiple words match).
- it knows about bodyparts that you can target certain actions (such as kick or pat) at.
- it can deal with object names that consist of multiple words (i.e. contain spaces). For instance, it understands when you type 'get the blue pill' when there are multiple pills on the table.
- tab-completion of commands on systems that support readline
- you can alter the meaning of a sentence by using words like fail, attempt, don't, suddenly, pretend
- you can put stuff into a bag and carry the bag, to avoid cluttering your inventory.
- you can refer to earlier used items and persons by using a pronoun ("examine box / drop it", "examine idiot / slap him").
- yelling something will actually be heard by creatures in adjacent locations. They'll get a message that someone is yelling something, and if possible, where the sound is coming from.
- text is nicely formatted when outputted (dynamically wrapped to a configurable width).
- uses ansi sequence to spice up the console output a bit (needs colorama on windows, falls back to plain text if not installed)
- uses smartypants to automatically render quotes, dashes, ellipsis in a nicer way.
- game can be saved (and reloaded)
- save game data is placed in the operating system's user data directory instead of some random location
- there's a list of 70+ creature races, adapted from the Dead Souls 2 mudlib

- supports two kinds of money: fantasy (gold/silver/copper) and modern (dollars). Text descriptions adapt to this.
- money can be given away, dropped on the floor, and picked up.
- it's possible for items to be combined into new items.
- game clock is independent of real-time wall clock, configurable speed and start time
- server 'tick' synced with command entry, or independent. This means things can happen in the background.
- there is a simple decorator that makes that a method gets invoked periodically, for asynchronous actions
- for more control you can make a 'deferred call' to schedule something to be called at a later time
- you can also quite easily schedule calls to be executed at a defined later moment in time
- using generators (yield statements) instead of regular input() calls, it is easy to create sequential dialogs (question-response) that will be handled without blocking the driver (the driver loop is not yet fully asynchronous but that may come in the future)
- easy definition of commands in separate functions, uses docstrings to define command help texts
- command function implementations are quite compact due to convenient parameters, and available methods on the game objects
- command code gets parse information from the soul parser as parameter; very little parsing needs to be done in the command code itself
- there's a large set of configurable parameters on a per-story basis
- stories can define their own introduction text and completion texts
- stories can define their own commands or override existing commands
- a lock/unlock/open/close door mechanism is provided with internal door codes to match keys (or key-like objects) against.
- action and event notification mechanism: objects are notified when things happen (such as the player entering a room, or someone saying a line of text) and can react on that.
- contains a simple virtual file system to provide easy resource loading / datafile storage.
- provides a simple pubsub/event signaling mechanism
- crashes are reported as detailed tracebacks showing local variable values per frame, to ease error reporting and debugging
- I/O abstraction layer to be able to create alternative interfaces to the engine
- for now, the game object model is object-oriented. You defined objects by instantiating prebuilt classes, or derive new classes from them with changed behavior. Currently this means that writing a game is very much a programming job. This may or may not improve in the future (to allow for more natural ways of writing a game story, in a DSL or whatever).
- a set of unit tests to validate a large part of the code

1.3 MUD mode versus Interactive Fiction mode

The Tale game driver launches in Interactive Fiction mode by default.

To run a story (or world, rather) in multi-user MUD mode, use the `--mode mud` command line switch. A whole lot of new commands and features are enabled when you do this (amongst others: message-of-the-day support and the 'stats' command). Running a IF story in MUD mode may cause some problems. Therefore you can specify in the story config what game modes your story supports.

1.4 Copyright

Tale is copyright © Irmen de Jong (irmen@razorvine.net | http://www.razorvine.net). Since version 3.4, it's licensed under GNU LGPL v3, see https://www.gnu.org/licenses/lgpl-3.0.html Versions older than that have a different license (GPL v3).

1.5 API documentation

Preliminary (auto-generated) API documentation:

1.5.1 Tale API

tale.accounts — Player account logic

Player account code.

class tale.accounts.**MudAccounts**(*databasefile: str*)
Handles the accounts (login, creation, etc) of mud users

Database: account(name, email, pw_hash, pw_salt, created, logged_in, locked) privilege(account, privilege) charstat(account, gender, stat1, stat2,...)

tale.author — Story Author tools

Utilities for story authors

tale.author.**do_zip**(*path: str*, *zipfilename: str*, *embed_tale: bool = False*, *verbose: bool = False*) → None
Zip a story (possibly including the tale library itself - but not its dependencies, to avoid license hassles) into a zip file.

tale.author.**run_from_cmdline**(*args: Sequence[str]*) → None
Entrypoint from the commandline to invoke the available tools from this module.

tale.base — Base classes

Mudlib base objects.

'Tale' mud driver, mudlib and interactive fiction framework Copyright by Irmen de Jong (irmen@razorvine.net)

object hierarchy:

```
MudObject (abstract base class, don't use directly)
  |
  +-- Location
  |
  +-- Item
  |     |
  |     +-- Weapon
  |     +-- Armour
  |     +-- Container
  |     +-- Key
  |
```

(continues on next page)

(continued from previous page)

```
  +-- Living (abstract base class, don't use directly)
  |      |
  |      +-- Player
  |      +-- NPC
  |           |
  |           +-- Shopkeeper
  |
  +-- Exit
         |
         +-- Door
```

Every object that can hold other objects does so in its "inventory" (a set). You can't access it directly, object.inventory returns a frozenset copy of it.

class tale.base.**MudObject**(*name: str*, *title: str = ''*, ***, *descr: str = ''*, *short_descr: str = ''*)

Root class of all objects in the mud world All objects have an identifying short name (will be lowercased), an optional short title (shown when listed in a room – don't use 'a' or 'the' or pronouns), and an optional longer description (shown when explicitly 'examined'). The long description is 'dedented' first, which means you can put it between triple-quoted-strings easily. Short_description is also optional, and is used in the text when a player 'looks' around. If it's not set, a generic 'look' message will be shown (something like "XYZ is here").

Extra descriptions (extra_desc) are used to make stuff more interesting and interactive Extra descriptions are accessed by players when they type `look at <thing>` where <thing> is any keyword you choose. For example, you might write a room description which includes the tantalizing sentence, `The wall looks strange here.` Using extra descriptions, players could then see additional detail by typing `look at wall.` There can be an unlimited number of Extra Descriptions.

add_extradesc(*keywords: Set[str]*, *description: str*) → None

For the set of keywords, add the extra description text

destroy(*ctx: Optional[tale.util.Context]*) → None

Common cleanup code that needs to be called when the object is destroyed

handle_verb(*parsed: tale.base.ParseResult*, *actor: tale.base.Living*) → bool

Handle a custom verb (specified in the verbs dict). Return True if handled, False if not handled.

init() → None

Secondary initialization/customization. Invoked after all required initialization has been done. You can easily override this in a subclass. It is not needed to call the MudObject super class init().

init_names(*name: str*, *title: str*, *descr: str*, *short_descr: str*) → None

(re)set the name and description attributes

notify_action(*parsed: tale.base.ParseResult*, *actor: tale.base.Living*) → None

Notify the object of an action performed by someone. This can be any verb, command, soul emote, custom verb. Uncompleted actions (error, or ActionRefused) are ignored. Custom verbs are notified however, even if they were already handled by handle_verb! It's good practice to first do a check like this:

```
if actor is self or parsed.verb in self.verbs:
    return  # avoid reacting to ourselves, or reacting to verbs we already␣
↪have a handler for
```

show_inventory(*actor: tale.base.Living*, *ctx: tale.util.Context*) → None

show the object's inventory to the actor

wiz_clone(*actor: tale.base.Living*) → tale.base.MudObject

clone the thing (performed by a wizard)

wiz_destroy(*actor: tale.base.Living*, *ctx: tale.util.Context*) → None
destroy the thing (performed by a wizard)

class tale.base.**Armour**(*name: str*, *title: str = ''*, *, *descr: str = ''*, *short_descr: str = ''*)
An item that can be worn by a Living (i.e. present in an armour itemslot)

class tale.base.**Container**(*name: str*, *title: str = ''*, *, *descr: str = ''*, *short_descr: str = ''*)
A bag-type container (i.e. an item that acts as a container) Allows insert and remove, and examine its contents, as opposed to an Item You can test for containment with 'in': item in bag

destroy(*ctx: Optional[tale.util.Context]*) → None
Common cleanup code that needs to be called when the object is destroyed

init() → None
Secondary initialization/customization. Invoked after all required initialization has been done. You can easily override this in a subclass. It is not needed to call the Item super class init().

init_inventory(*items: Iterable[tale.base.Item]*) → None
Set the container's initial inventory

class tale.base.**Door**(*directions: Sequence[str], target_location: Union[str, tale.base.Location], short_descr: str, long_descr: str = '', *, enter_msg: str = '', locked: bool = False, opened: bool = False, key_code: str = ''*)
A special exit that connects one location to another but which can be closed or even locked. Because a single door is still only one-way, you have to create a second -linked- door to go back. This is easily done by the `reverse_door` method.

allow_passage(*actor: tale.base.Living*) → None
Is the actor allowed to move through this door?

check_key(*item: tale.base.Item*) → bool
Check if the item is a proper key for this door (based on key_code)

close(*actor: tale.base.Living*, *item: tale.base.Item = None*) → None
Close the door with optional item. Notifies actor and room of this event.

classmethod connect(*from_loc: tale.base.Location, directions: Sequence[str], short_descr: str, long_descr: str, to_loc: tale.base.Location, return_directions: Sequence[str], return_short_descr: str, return_long_descr: str, locked: bool = False, opened: bool = False, key_code: str = ''*) → Tuple[tale.base.Door, tale.base.Door]
Create a pair of doors that connect two locations. (This requires two door definitions because the directions and descriptions differ for the to- and return-exists)

insert(*item: Union[tale.base.Living, tale.base.Item], actor: Optional[tale.base.Living]*) → None
used when the player tries to put a key into the door, for instance.

lock(*actor: tale.base.Living*, *item: tale.base.Item = None*) → None
Lock the door with the proper key (optional).

open(*actor: tale.base.Living*, *item: tale.base.Item = None*) → None
Open the door with optional item. Notifies actor and room of this event.

reverse_door(*directions: Sequence[str], returning_location: tale.base.Location, short_description: str, long_description: str = ''*) → tale.base.Door
Set up a second door in the other location that is paired with this door. Opening this door will also open the other door etc. Returns the new door object. (we need 2 doors because the name/exit descriptions are often different from both locations)

search_key(*actor: tale.base.Living*) → Optional[tale.base.Item]
Does the actor have a proper key? Return the item if so, otherwise return None.

unlock(*actor: tale.base.Living*, *item: tale.base.Item = None*) → None
Unlock the door with the proper key (optional).

class tale.base.**Exit**(*directions: Sequence[str], target_location: Union[str, tale.base.Location], short_descr: str, long_descr: str = '', *, enter_msg: str = ''*)
An 'exit' that connects one location to another. It is strictly one-way! Directions can be a single string or a sequence of directions (all meaning the same exit). You can use a Location object as target, or a string designating the location (for instance "town.square" means the square location object in game.zones.town). If using a string, it will be retrieved and bound at runtime. Short_description will be shown when the player looks around the room. Long_description is optional and will be shown instead if the player examines the exit. Enter_msg is the text shown to the player when they succesfully enter/pass through the exit/door. The exit's direction is stored as its name attribute (if more than one, the rest are aliases). Note that the exit's origin is not stored in the exit object.

allow_passage(*actor: tale.base.Living*) → None
Is the actor allowed to move through the exit? Raise ActionRefused if not

bind(*location: tale.base.Location*) → None
Binds the exit to a location.

classmethod connect(*from_loc: tale.base.Location, directions: Sequence[str], short_descr: str, long_descr: str, to_loc: tale.base.Location, return_directions: Sequence[str], return_short_descr: str, return_long_descr: str*) → Tuple[tale.base.Exit, tale.base.Exit]
Create a pair of exits that connect two locations. (This requires two exit definitions because the directions and descriptions differ for the to- and return-exists)

names
a list of all the names of this direction (name followed by aliases)

class tale.base.**Item**(*name: str*, *title: str = ''*, *, *descr: str = ''*, *short_descr: str = ''*)
Root class of all Items in the mud world. Items are physical objects. Items can usually be moved, carried, or put inside other items. They have a name and optional short and longer descriptions. Regular items cannot contain other things, so it makes to sense to check containment.

allow_item_move(*actor: Optional[Living], verb: str = 'move'*) → None
Does the item allow to be moved (picked up, given away) by someone? (yes; no ActionRefused is raised)

clone() → tale.base.Item
Create a copy of an existing Item. Only allowed when it has an empty inventory (to avoid problems). Caller has to make sure the resulting copy is moved to its proper destination location.

combine(*other: List[Item], actor: tale.base.Living*) → Optional[tale.base.Item]
Combine the other thing(s) with us. If successful, return the new Item to replace us + all other items with. (so 'other' must NOT contain any item not used in combining the things, or it will be silently lost!) If stuff cannot be combined, return None (or raise an ActionRefused with a particular message).

init() → None
Secondary initialization/customization. Invoked after all required initialization has been done. You can easily override this in a subclass. It is not needed to call the Item super class init().

move(*target: Union[Location, Container, Living], actor: Optional[tale.base.Living] = None, *, silent: bool = False, is_player: bool = False, verb: str = 'move', direction_names: Sequence[str] = None*) → None
Leave the container the item is currently in, enter the target container (transactional). Because items can move on various occasions, there's no message being printed. The silent and is_player arguments are not used when moving items – they're used for the movement of livings.

notify_moved(*source_container: Union[Location, Container, Living], target_container: Union[Location, Container, Living], actor: Optional[Living]*) → None
Called when the item has been moved from one place to another

static search_item(*name: str, collection: Iterable[Item]*) → Optional[tale.base.Item]
Searches an item (by name) in a collection of Items. Returns the first match (or None if nothing found). Also considers aliases and titles.

show_inventory(*actor: tale.base.Living*, *ctx: tale.util.Context*) → None
show the object's contents to the actor

wiz_clone(*actor: Living*, *make_clone: bool = True*) → Item
clone the thing (performed by a wizard)

wiz_destroy(*actor: Living*, *ctx: tale.util.Context*) → None
destroy the thing (performed by a wizard)

class tale.base.**Living**(*name: str*, *gender: str*, *, *race: str = 'human'*, *title: str = ''*, *descr: str = ''*, *short_descr: str = ''*)
A living entity in the mud world (also known as an NPC). Livings sometimes have a heart beat 'tick' that makes them interact with the world. They are always inside a Location (Limbo when not specified yet). They also have an inventory object, and you can test for containment with item in living.

allow_give_item(*item: tale.base.Item, actor: Optional[Living]*) → None
Do we accept given items? Raise ActionRefused if not.

allow_give_money(*amount: float, actor: Optional[Living]*) → None
Do we accept money? Raise ActionRefused if not.

destroy(*ctx: Optional[tale.util.Context]*) → None
Common cleanup code that needs to be called when the object is destroyed

do_command_verb(*cmdline: str*, *ctx: tale.util.Context*) → None
Perform a verb, parsed from a command line. This is an easy way to make a Npc do something, but it has a pretty large performance overhead. If you can, you should use low level methods instead (such as `tell_others` or `do_socialize` etc) The verb can be a soul verb (such as 'ponder') but also a command verb. Custom dynamic verbs added by the environment are not supported (yet), and neither are commands that initiate a dialog (generators) This function is not used in the processing of player commands!

do_forced_cmd(*actor: Living*, *parsed: tale.base.ParseResult*, *ctx: tale.util.Context*) → None
Perform a (pre-parsed) command because the actor forced us to do it.

This code is fairly similar to the __process_player_command from the driver but it doesn't deal with as many error situations, and just bails out if it gets confused. It does try its best to support the following: - custom location verbs (such as 'sell' in a shop) - exit handling - built-in cmds (such as 'drop'/'take') Note that soul emotes are handled by do_socialize_cmd instead.

do_socialize(*cmdline: str*, *external_verbs: Set[str] = set()*) → None
Perform a command line with a socialize/soul verb on the living's behalf. It only performs soul emotes, no custom command functions!

do_socialize_cmd(*parsed: tale.base.ParseResult*) → None
A soul verb such as 'ponder' was entered. Socialize with the environment to handle this. Some verbs may trigger a response or action from something or someone else.

get_wiretap() → tale.pubsub.Topic
get a wiretap for this living

handle_verb(*parsed: tale.base.ParseResult*, *actor: tale.base.Living*) → bool
Handle a custom verb (specified in the verbs dict). Return True if handled, False if not handled.

init_gender(*gender: str*) → None
(re)set gender attributes

init_inventory(*items: Iterable[tale.base.Item]*) → None
Set the living's initial inventory

insert(*item: Union[Living, tale.base.Item], actor: Optional[Living]*) → None
Add an item to the inventory.

locate_item(*name: str*, *include_inventory: bool = True*, *include_location: bool = True*, *include_containers_in_inventory: bool = True*) → Tuple[Optional[tale.base.Item], Union[tale.base.Location, tale.base.Container, tale.base.Living, None]]
Searches an item within the 'visible' world around the living including his inventory. If there's more than one hit, just return the first. Returns (None,None) or (item, containing_object)

look(*short: Optional[bool] = None*) → None
look around in your surroundings. Dummy for base livings (they don't perform 'look' nor react to it).

move(*target: Union[tale.base.Location, Container, Living], actor: Optional[tale.base.Living] = None, *, silent: bool = False, is_player: bool = False, verb: str = 'move', direction_names: Sequence[str] = None*) → None
Leave the current location, enter the new location (transactional). Moving a living is only supported to a Location target. Messages are being printed to the locations if the move was successful.

notify_action(*parsed: tale.base.ParseResult*, *actor: tale.base.Living*) → None
Notify the living of an action performed by someone. This can be any verb, command, soul emote, custom verb. Uncompleted actions (error, or ActionRefused) are ignored. Custom verbs are notified however, even if they were already handled by handle_verb! It's good practice to first do a check like this:

```
if actor is self or parsed.verb in self.verbs:
    return  # avoid reacting to ourselves, or reacting to verbs we already␣
↪have a handler for
```

parse(*commandline: str*, *external_verbs: Set[str] = set()*) → tale.base.ParseResult
Parse the commandline into something that can be processed by the soul (ParseResult)

remember_previous_parse() → None
remember the previously parsed data, soul uses this to reference back to earlier items/livings

remove(*item: Union[Living, tale.base.Item], actor: Optional[Living]*) → None
remove an item from the inventory

search_item(*name: str*, *include_inventory: bool = True*, *include_location: bool = True*, *include_containers_in_inventory: bool = True*) → Optional[tale.base.Item]
The same as locate_item except it only returns the item, or None.

select_random_move() → Optional[tale.base.Exit]
Select a random accessible exit to move to. Avoids exits to a room that have no exits (traps). If no suitable exit is found in a few random attempts, return None.

show_inventory(*actor: tale.base.Living*, *ctx: tale.util.Context*) → None
show the living's inventory to the actor

start_attack(*victim: tale.base.Living*) → None
Starts attacking the given living until death ensues on either side.

tell(*message: str*, *, *end: bool = False*, *format: bool = True*) → tale.base.Living
Every living thing in the mud can receive an action message. Message will be converted to str if required. For players this is usually printed to their screen, but for all other livings the default is to do nothing – except for making sure that the message is sent to any wiretaps that may be present. The Living could react on the message, but this is not advisable because you'll have to parse the string again to figure out what happened... (there are better ways to react on stuff that happened). The Living itself is returned so you can easily chain calls. Note: end and format parameters are ignored for Livings but may be useful when this function is called on a subclass such as Player.

tell_later (*message: str*) → None
Tell something to this creature, but do it after all other messages.

tell_others (*message: str*, *target: Optional[Living] = None*) → None
Send a message to the other livings in the location, but not to self. There are a few formatting strings for easy shorthands: {actor}/{Actor} = the acting living's title / acting living's title capitalized (subject in the sentence) {target}/{Target} = the target's title / target's title capitalized (object in the sentence) If you need even more tweaks with telling stuff, use living.location.tell directly.

validate_socialize_targets (*parsed: tale.base.ParseResult*) → None
check if any of the targeted objects is an exit

wiz_clone (*actor: Living*, *make_clone: bool = True*) → Living
clone the thing (performed by a wizard)

wiz_destroy (*actor: Living*, *ctx: tale.util.Context*) → None
destroy the thing (performed by a wizard)

class tale.base.**Location** (*name: str*, *descr: str = ''*)
A location in the mud world. Livings and Items are in it. Has connections ('exits') to other Locations. You can test for containment with 'in': item in loc, npc in loc

add_exits (*exits: Iterable[Exit]*) → None
Adds every exit from the sequence as an exit to this room.

destroy (*ctx: Optional[tale.util.Context]*) → None
Common cleanup code that needs to be called when the object is destroyed

get_wiretap () → tale.pubsub.Topic
get a wiretap for this location

handle_verb (*parsed: tale.base.ParseResult*, *actor: tale.base.Living*) → bool
Handle a custom verb (specified in the verbs dict of a living/item/exit in this location). Return True if handled, False if not handled.

init_inventory (*objects: Iterable[Union[tale.base.Item, Living]]*) → None
Set the location's initial item and livings 'inventory'

insert (*obj: Union[Living, tale.base.Item]*, *actor: Optional[Living]*) → None
Add item to the contents of the location (either a Living or an Item)

look (*exclude_living: Optional[tale.base.Living] = None*, *short: bool = False*) → Sequence[str]
returns a list of paragraph strings describing the surroundings, possibly excluding one living from the description list

message_nearby_locations (*message: str*) → None
Tells a message to adjacent locations, where adjacent is defined by being connected via an exit. If the adjacent location has an obvious returning exit to the source location (via one of the most obvious routes n/e/s/w/up/down/etc.), it hen also get information on what direction the sound originated from. This is used for loud noises such as yells!

nearby (*no_traps: bool = True*) → Iterable[Location]
Returns a sequence of all adjacent locations, normally avoiding 'traps' (locations without a way back). (this may be expanded in the future with a way to search further than just 1 step away)

notify_action (*parsed: tale.base.ParseResult*, *actor: tale.base.Living*) → None
Notify the location, the items in it, and the livings in it, of an action performed by someone. This can be any verb, command, soul emote, custom verb. Uncompleted actions (error, or ActionRefused) are ignored. Custom verbs are notified however, even if they were already handled by handle_verb! It's good practice to first do a check like this:

```
if actor is self or parsed.verb in self.verbs:
    return  # avoid reacting to ourselves, or reacting to verbs we already␣
↪have a handler for
```

notify_npc_arrived(*npc: tale.base.Living*, *previous_location: tale.base.Location*) → None
A NPC has arrived in this location. When you override this be sure to call base method. This event is not delegated to all items or creatures in the location! If you need that, you should create a pubsub topic event, where the correct objects are listening on.

notify_npc_left(*npc: tale.base.Living*, *target_location: tale.base.Location*) → None
A NPC has left the location. When you override this be sure to call base method. This event is not delegated to all items or creatures in the location! If you need that, you should create a pubsub topic event, where the correct objects are listening on.

notify_player_arrived(*player*, *previous_location: tale.base.Location*) → None
A player has arrived in this location. When you override this be sure to call base method. This event is not delegated to all items or creatures in the location! If you need that, you should create a pubsub topic event, where the correct objects are listening on.

notify_player_left(*player*, *target_location: tale.base.Location*) → None
A player has left this location. When you override this be sure to call base method. This event is not delegated to all items or creatures in the location! If you need that, you should create a pubsub topic event, where the correct objects are listening on.

remove(*obj: Union[Living, tale.base.Item]*, *actor: Optional[Living]*) → None
Remove obj from this location (either a Living or an Item)

search_living(*name: str*) → Optional[tale.base.Living]
Search for a living in this location by its name (and title, if no names match). Is alias-aware. If there's more than one match, returns the first. None if nothing found.

tell(*room_msg: str*, *exclude_living: Optional[tale.base.Living] = None*, *specific_targets: Set[Union[Living, Item, Exit]] = None*, *specific_target_msg: str = ''*) → None
Tells something to the livings in the room (excluding the living from exclude_living). This is just the message string! If you want to react on events, consider not doing that based on this message string. That will make it quite hard because you need to parse the string again to figure out what happened... Use handle_verb / notify_action instead.

class tale.base.**Weapon**(*name: str*, *title: str = ''*, *, *descr: str = ''*, *short_descr: str = ''*)
An item that can be wielded by a Living (i.e. present in a weapon itemslot), and that can be used to attack another Living.

class tale.base.**Key**(*name: str*, *title: str = ''*, *, *descr: str = ''*, *short_descr: str = ''*)
A key which has a unique code. It can be used to open a matching Door. Set the door or code using the key_for method.

init() → None
Secondary initialization/customization. Invoked after all required initialization has been done. You can easily override this in a subclass. It is not needed to call the Item super class init().

key_for(*door: tale.base.Door = None*, *code: str = ''*) → None
Makes this key a key for the given door. (basically just copies the door's key_code)

class tale.base.**Soul**
The 'soul' of a Living (most importantly, a Player). Handles the high level verb actions and allows for social player interaction. Verbs that actually do something in the environment (not purely social messages) are implemented elsewhere.

check_name_with_spaces(*words: Sequence[str], startindex: int, all_livings: Dict[str, tale.base.Living], all_items: Dict[str, tale.base.Item], all_exits: Dict[str, tale.base.Exit]*) → Tuple[Union[tale.base.Living, tale.base.Item, tale.base.Exit, None], str, int]

Searches for a name used in sentence where the name consists of multiple words (separated by space). You provide the sequence of words that forms the sentence and the startindex of the first word to start searching. Searching is done in the livings, items, and exits dictionaries, in that order. The name being searched for is gradually extended with more words until a match is found. The return tuple is (matched_object, matched_name, number of words used in match). If nothing is found, a tuple (None, None, 0) is returned.

match_previously_parsed(*player: tale.base.Living*, *pronoun: str*) → List[Tuple[Any, str]]

Try to connect the pronoun (it, him, her, them) to a previously parsed item/living. Returns a list of (who, replacement-name) tuples. The reason we return a replacement-name is that the parser can replace the pronoun by the proper name that would otherwise have been used in that place.

parse(*player: tale.base.Living*, *cmd: str*, *external_verbs: Set[str] = set()*) → tale.base.ParseResult

Parse a command string, returns a ParseResult object.

static poss_replacement(*actor: tale.base.Living, target: Optional[tale.base.MudObject], observer: Optional[tale.base.Living]*) → str

determines what word to use for a POSS

process_verb(*player: tale.base.Living*, *commandstring: str*, *external_verbs: Set[str] = set()*) → Tuple[str, Tuple[Set[Union[Living, Item, Exit]], str, str, str]]

Parse a command string and return a tuple containing the main verb (tickle, ponder, . . .) and another tuple containing the targets of the action (excluding the player) and the various action messages. Any action qualifier is added to the verb string if it is present ("fail kick").

process_verb_parsed(*player: tale.base.Living*, *parsed: tale.base.ParseResult*) → Tuple[Set[Union[Living, Item, Exit]], str, str, str]

This function takes a verb and the arguments given by the user, creates various display messages that can be sent to the players and room, and returns a tuple: (targets-without-player, playermessage, roommessage, targetmessage) Target can be a Living, an Item or an Exit.

spacify(*string: str*) → str

returns string prefixed with a space, if it has contents. If it is empty, prefix nothing

who_replacement(*actor: tale.base.Living, target: tale.base.MudObject, observer: Optional[tale.base.Living]*) → str

determines what word to use for a WHO

tale.charbuilder — Character builder

Character builder for multi-user mode.

class tale.charbuilder.**IFCharacterBuilder**(*conn: tale.player.PlayerConnection*, *config: tale.story.StoryConfig*)

Create a new player character interactively.

class tale.charbuilder.**MudCharacterBuilder**(*conn: tale.player.PlayerConnection*, *name: str*, *config: tale.story.StoryConfig*)

Create a new player character interactively.

tale.driver — Game driver/server common logic

Mud driver (server).

class tale.driver.**Commands**

Some utility functions to manage the registered commands.

class tale.driver.**Deferred**(*due_gametime: datetime.datetime, action: Callable, vargs: Sequence[Any], kwargs: Dict[str, Any], *, periodical: Tuple[float, float] = None*)

Represents a callable action that will be invoked (with the given arguments) sometime in the future. This object captures the action that must be invoked in a way that is serializable. That means that you can't pass all types of callables, there are a few that are not serializable (lambda's and scoped functions). They will trigger an error if you use those. If you set a (low_seconds, high_seconds) periodical tuple, the deferred will be called periodically where the next trigger time is randomized within the given interval. The due time is given in Game Time, not in real/wall time! Note that the vargs/kwargs should be serializable or savegames are impossible!

when_due(*game_clock: tale.util.GameDateTime*, *realtime: bool = False*) → datetime.timedelta

In what time is this deferred due to occur? (timedelta) Normally it is in terms of game-time, but if you pass realtime=True, you will get the real-time timedelta.

class tale.driver.**Driver**

The Mud 'driver'. Reads story file and config, initializes game state. Handles main game loop, player connections, and loading/saving of game state.

DeferDueType

alias of `typing.Union`

current_custom_verbs(*player: tale.player.Player*) → Dict[str, str]

returns dict of the currently recognised custom verbs (verb->helptext mapping)

current_verbs(*player: tale.player.Player*) → Dict[str, str]

return a dict of all currently recognised verbs, and their help text

defer(*due: Union[datetime.datetime, float, Tuple[float, float, float]], action: Callable, *vargs, **kwargs*) → tale.driver.Deferred

Register a deferred callable action (optionally with arguments). The vargs and the kwargs all must be serializable. Note that the due time can be one of: - datetime.datetime *in game time* (not real time!) when the deferred should trigger. - float, meaning the number of real-time seconds after the current time (minimum: 0.1 sec) - tuple(initial_secs, low_secs, high_secs), meaning it is periodical within the given time interval. The deferred gets a kwarg 'ctx' set to a Context object, if it has a 'ctx' argument in its signature. (If not, that's okay too) Receiving the context is often useful, for instance you can register a new deferred on the ctx.driver without having to access a global driver object. Triggering a deferred can not occur sooner than the server tick period!

pubsub_event(*topicname: Union[str, Tuple], event: Union[Callable, Tuple[tale.player.PlayerConnection, str]]*) → None

override this event receive method in a subclass

search_player(*name: str*) → Optional[tale.player.Player]

Look through all the logged in players for one with the given name. Returns None if no one is known with that name.

start(*game_file_or_path: str*) → None

Start the driver from a parsed set of arguments

uptime

gives the server uptime in a (hours, minutes, seconds) tuple

tale.driver_if — IF single player Game driver

Single user driver (for interactive fiction).

class tale.driver_if.**IFDriver**(*, *screen_delay: int = 40*, *gui: bool = False*, *web: bool = False*, *wizard_override: bool = False*)

The Single user 'driver'. Used to control interactive fiction where there's only one 'player'.

main_loop(*conn: Optional[tale.player.PlayerConnection]*) → None
The game loop, for the single player Interactive Fiction game mode. Until the game is exited, it processes player input, and prints the resulting output.

tale.driver_mud — MUD multiplayer Game driver/server

Mud driver (multi user server).

class tale.driver_mud.**LimboReaper**
The Grim Reaper hangs about in Limbo, and makes sure no one stays there for too long.

notify_action(*parsed: tale.base.ParseResult*, *actor: tale.base.Living*) → None
Notify the living of an action performed by someone. This can be any verb, command, soul emote, custom verb. Uncompleted actions (error, or ActionRefused) are ignored. Custom verbs are notified however, even if they were already handled by handle_verb! It's good practice to first do a check like this:

```
if actor is self or parsed.verb in self.verbs:
    return  # avoid reacting to ourselves, or reacting to verbs we already␣
↪have a handler for
```

class tale.driver_mud.**MudDriver**(*restricted=False*)
The Mud 'driver'. Multi-user server variant of the single player Driver.

main_loop(*conn: Optional[tale.player.PlayerConnection]*) → None
The game loop, for the multiplayer MUD mode. Until the server is shut down, it processes player input, and prints the resulting output.

show_motd(*player: tale.player.Player*, *notify_no_motd: bool = False*) → None
Prints the Message-Of-The-Day file, if present.

tale.errors — Exceptions

Exception classes

exception tale.errors.**ActionRefused**
The action that was tried was refused by the situation or target object

exception tale.errors.**AsyncDialog**(*dialog: Generator[[Tuple[str, Any], str], None]*)
Command execution needs to continue with the async dialog generator given as argument.

exception tale.errors.**LocationIntegrityError**(*msg: str*, *direction: str*, *exit: Any*, *location: Any*)
When the driver notices an integrity problem with locations, exits, etc.

exception tale.errors.**NonSoulVerb**(*parseresult*)
The soul's parser encountered a verb that cannot be handled by the soul itself. However the command string has been parsed and the calling code could try to handle the verb by itself instead.

exception tale.errors.**ParseError**
Problem with parsing the user input. Should be shown to the user as a nice error message.

exception tale.errors.**RetryParse**(*command: str*)
Retry the command as a different one

exception tale.errors.**RetrySoulVerb**
Retry a command as soul verb instead.

exception tale.errors.**SecurityViolation**
Some security constraint was violated

exception tale.errors.**SessionExit**
Player session ends.

exception tale.errors.**StoryCompleted**
This is raised as soon as the (IF) story has been completed by the player! Can be successful, or failed ending. You'll have to print the correct message yourself. Do not use this in a Mud story.

exception tale.errors.**StoryConfigError**
There was a problem with the story configuration

exception tale.errors.**TaleError**
base class for tale related errors

exception tale.errors.**TaleFlowControlException**
base class for flow-control exceptions

exception tale.errors.**UnknownVerbException**(*verb: str, words: Sequence[str], qualifier: str*)
The soul doesn't recognise the verb that the user typed. The engine can and should search for other places that define this verb first. If nothing recognises it, this error should be shown to the user in a nice way.

tale.lang — Language utilities

Language processing related operations.

tale.lang.**A**(*word: str*) → str
prefix an article 'A' or 'An' capitalized. (if possible)

class tale.lang.**OrderedCounter**(***kwds*)
A counter that remembers the order in which things are being counted.

classmethod fromkeys(*S*[, *v*]) → New ordered dictionary with keys from S.
If not specified, the value defaults to None.

tale.lang.**a**(*noun_phrase: str*) → str
prefix an article 'a' or 'an' (if possible)

tale.lang.**adverb_by_prefix**(*prefix: str*, *amount: int = 5*) → List[str]
Return a list of adverbs starting with the given prefix, up to the given amount Uses binary search in the sorted adverbs list, O(log n)

tale.lang.**fullstop**(*sentence: str*, *punct: str = '.'*) → str
adds a fullstop to the end of a sentence if needed

tale.lang.**fullverb**(*verb: str*) → str
return the full verb: shoot->shooting, poke->poking

tale.lang.**join**(*words: Iterable[str], conj: str = 'and', group_multi: bool = True*) → str
Join a list of words to 'a,b,c, and e' If a word occurs multiple times (and group_multi=True), show 'thing and thing' as 'two things' instead.

tale.lang.**ordinal**(*number: int*) → str
return the simple ordinal (1st, 3rd, 8th etc) of a number. Supports positive and negative ints.

tale.lang.**spell_number**(*number: float*) → str
Return a spelling of the number. Supports positive and negative ints, floats, and recognises popular fractions such as 0.5 and 0.25. Numbers that are very near a whole number are also returned as "about N". Any fraction that can not be spelled out (or is larger than +/- 100) will not be spelled out in words, but returned in numerical form.

tale.lang.**spell_ordinal**(*number: int*) → str
Return a spelling of the ordinal number. Supports positive and negative ints.

tale.lang.**split**(*string: str*) → List[str]
Split a string on whitespace, but keeps words enclosed in quotes (‘ or “) together. The quotes themselves are stripped out.

tale.main — Command line entrypoint

Main startup class

tale.main.**run_from_cmdline**(*cmdline: Sequence[str]*) → None
Run Tale from the commandline.

tale.player — Players

Player code

class tale.player.**Player**(*name: str*, *gender: str*, *, *race: str = 'human'*, *descr: str = ''*, *short_descr: str = ''*)
Player controlled entity. Has a Soul for social interaction.

allow_give_item(*item: tale.base.Item, actor: Optional[tale.base.Living]*) → None
Do we accept given items? Raise ActionRefused if not. For Player, the default is that we accept.

allow_give_money(*amount: float, actor: Optional[tale.base.Living]*) → None
Do we accept money? Raise ActionRefused if not. For Player, the default is that we accept.

destroy(*ctx: Optional[tale.util.Context]*) → None
Common cleanup code that needs to be called when the object is destroyed

get_pending_input() → Sequence[str]
return the full set of lines in the input buffer (if any)

init_names(*name: str*, *title: str*, *descr: str*, *short_descr: str*) → None
(re)set the name and description attributes

look(*short: Optional[bool] = None*) → None
look around in your surroundings (it excludes the player himself from livings)

move(*target: Union[Location, Container, Living], actor: tale.base.Living = None, *, silent: bool = False, is_player: bool = True, verb: str = 'move', direction_names: Sequence[str] = None*) → None
Delegate to Living but with is_player set to True. Moving the player is only supported to a target Location.

pubsub_event(*topicname: Union[str, Tuple], event: Tuple[tale.base.MudObject, str]*) → None
override this event receive method in a subclass

search_extradesc(*keyword: str*, *include_inventory: bool = True*, *include_containers_in_inventory: bool = False*) → str
Searches the extradesc keywords for an location/living/item within the ‘visible’ world around the player, including their inventory. If there’s more than one hit, just return the first extradesc description text.

store_input_line(*cmd: str*) → None
store a line of entered text in the input command buffer

tell(*message: str*, *, *end: bool = False*, *format: bool = True*) → tale.base.Living
Sends a message to a player, meant to be printed on the screen. Message will be converted to str if required. If you want to output a paragraph separator, either set end=True or tell a single newline. If you provide format=False, this paragraph of text won’t be formatted when it is outputted, and whitespace is untouched. Empty strings aren’t outputted at all. The player object is returned so you can chain calls.

tell_object_location(*obj: tale.base.MudObject, known_container: Union[tale.base.Living, tale.base.Item, tale.base.Location, None], print_parentheses: bool = True*) → None
Tells the player some details about the location of the given object.

tell_text_file(*file_resource: tale.vfs.Resource*, *reformat=True*) → None
Show the contents of the given text file resource to the player.

test_get_output_paragraphs() → Sequence[Sequence[str]]
Gets the accumulated output paragraphs in raw form. This is for test purposes. No text styles are included.

test_peek_output_paragraphs() → Sequence[Sequence[str]]
Returns a copy of the output paragraphs that sit in the buffer so far This is for test purposes. No text styles are included.

class tale.player.**PlayerConnection**(*player: tale.player.Player = <Player 'dummy' #2 @ 0x7f06dc6bcc50, privs:->, io: tale.tio.iobase.IoAdapterBase = <tale.tio.iobase.IoAdapterBase object>*)
Represents a player and the i/o connection that is used for him/her. Provides high level i/o operations to input commands and write output for the player. Other code should not have to call the i/o adapter directly.

get_output() → str
Gets the accumulated output lines, formats them nicely, and clears the buffer. If there is nothing to be outputted, empty string is returned.

input_direct(*prompt: str*) → str
Writes any pending output and prompts for input directly. Returns stripped result. The driver does NOT use this for the regular game loop! This call is *blocking* and will not work in a multi user situation.

output(**lines*) → None
directly writes the given text to the player's screen, without buffering and formatting/wrapping

output_no_newline(*line: str*) → None
similar to output() but writes a single line, without newline at the end

write_output() → None
print any buffered output to the player's screen

class tale.player.**TextBuffer**
Buffered output for the text that the player will see on the screen. The buffer queues up output text into paragraphs. Notice that no actual output formatting is done here, that is performed elsewhere.

p() → None
Paragraph terminator. Start new paragraph on next line.

print(*line: str*, *end: bool = False*, *format: bool = True*) → None
Write a line of text. A single space is inserted between lines, if format=True. If end=True, the current paragraph is ended and a new one begins. If format=True, the text will be formatted when output, otherwise it is outputted as-is.

tale.pubsub — Simple synchronous pubsub/event mechanism

Simple Pubsub signaling. Provides immediate (synchronous) sending, or store-and-forward sending when the sync() function is called. Uses weakrefs to not needlessly lock subscribers/topics in memory.

'Tale' mud driver, mudlib and interactive fiction framework Copyright by Irmen de Jong (irmen@razorvine.net)

Currently defined pubsub topics used by the Tale driver:

"driver-pending-actions" Events are callables to be executed in the server tick loop. You can subscribe but only the driver may execute the events.

"driver-pending-tells" Tells (messages) that have to be delivered to actors, after any other messages have been processed. You can subscribe but only the driver may execute the events.

"driver-async-dialogs" actions that kick off new async dialogs (generators). You can subscribe but only the driver may execute the events.

("wiretap-location", <location name>) Used by the wiretapper on a location

tale.pubsub.**topic**(*name: Union[str, Tuple]*) → tale.pubsub.Topic
Create a topic object (singleton). Name can be a string or a tuple.

tale.pubsub.**unsubscribe_all**(*subscriber: tale.pubsub.Listener*) → None
unsubscribe the given subscriber object from all topics that it may have been subscribed to.

class tale.pubsub.**Listener**
Base class for all pubsub listeners (subscribers)

exception NotYet
raise this from pubsub_event to signal that you don't want to consume the event just yet

pubsub_event(*topicname: Union[str, Tuple], event: Any*) → Any
override this event receive method in a subclass

tale.races — Races and creature attributes

Race definitions. Races adapted from Dead Souls 2 mudlib (a superset of the races from Nightmare mudlib).

class tale.races.**BodySize**(*text*, *order*)
An enumeration.

class tale.races.**BodyType**
An enumeration.

class tale.races.**Flags**(*flying*, *limbless*, *nonbiting*, *swimming*, *nonmeat*, *playable*)

flying
Alias for field number 0

limbless
Alias for field number 1

nonbiting
Alias for field number 2

nonmeat
Alias for field number 4

playable
Alias for field number 5

swimming
Alias for field number 3

class tale.races.**Race**(*name*, *body*, *language*, *mass*, *size*, *flags*)

body
Alias for field number 1

flags
Alias for field number 5

language
Alias for field number 2

mass
Alias for field number 3

name
Alias for field number 0

size
Alias for field number 4

tale.savegames — Save/Load game logic

tale.savegames.**mudobj_ref**(*mudobj: tale.base.MudObject*) → Optional[Tuple[int, str, str, str]]
generate a serializable reference (vnum, name, classname, baseclassname) for a MudObject

tale.shop — Shops

Shopping and shopkeepers.

'Tale' mud driver, mudlib and interactive fiction framework Copyright by Irmen de Jong (irmen@razorvine.net)

Shopping related commands will be roughly:

```
SHOP/LIST [item type]
    list what the shop has for sale
INFO/INQUIRE/ASK about [item/number]
    same as "ask [shopkeeper] about [item/number]"
    It will display info about the item on sale, as if you examined it.
BUY
  > buy sword         (buy the first sword on the list)
  > buy #3            (buy the third item on the list)
SELL
  > sell sword        (sell the first sword in your inventory)
VALUE/APPRAISE
```

class tale.shop.**ShopBehavior**
the data describing the behavior of a particular shop

class tale.shop.**Shopkeeper**(*name: str*, *gender: str*, ***, *race: str = 'human'*, *title: str = ''*, *descr: str = ''*, *short_descr: str = ''*)

allow_give_item(*item: tale.base.Item, actor: Optional[Living]*) → None
Do we accept given items? Raise ActionRefused if not. Shopkeeper can only be sold items to!

handle_verb(*parsed: tale.base.ParseResult*, *actor: tale.base.Living*) → bool
Handle a custom verb (specified in the verbs dict). Return True if handled, False if not handled.

init() → None
Secondary initialization/customization. Invoked after all required initialization has been done. You can easily override this in a subclass. It is not needed to call the MudObject super class init().

notify_action(*parsed: tale.base.ParseResult*, *actor: tale.base.Living*) → None
Notify the living of an action performed by someone. This can be any verb, command, soul emote, custom

verb. Uncompleted actions (error, or ActionRefused) are ignored. Custom verbs are notified however, even if they were already handled by handle_verb! It's good practice to first do a check like this:

```
if actor is self or parsed.verb in self.verbs:
    return  # avoid reacting to ourselves, or reacting to verbs we already␣
↪have a handler for
```

tale.story — Story configuration

Story configuration and base classes to create your own story with.

class tale.story.**TickMethod**
An enumeration.

class tale.story.**GameMode**
An enumeration.

class tale.story.**MoneyType**
An enumeration.

class tale.story.**StoryBase**
base class for tale story classes.

create_account_dialog(*playerconnection*, *playernaming*) → Generator
Override to add extra dialog options to the character creation process. Because there's no actual player yet, you receive PlayerConnection and PlayerNaming arguments. Write stuff to the user via playerconnection.output(...) Ask questions using the yield "input", "question?" mechanism. Return True to declare all is well, and False to abort the player creation process.

goodbye(*player*) → None
goodbye text when player quits the game

init(*driver*) → None
Called by the game driver when it is done with its initial initialization.

init_player(*player*) → None
Called by the game driver when it has created the player object (after successful login). You can set the hint texts on the player object, or change the state object, etc. For an IF game there is only one player. For a MUD game there will be many players, and every player that logs in can be further initialized here.

welcome(*player*) → str
Welcome text when player enters a new game If you return a non-empty string, it is used as an input prompt before continuing (a pause).

welcome_savegame(*player*) → str
Welcome text when player enters the game after loading a saved game If you return a non-empty string, it is used as an input prompt before continuing (a pause).

class tale.story.**StoryConfig**
Story configuration settings. The reason this is in a separate class, is that these settings are all simple values and are serializable, so they can be saved to disk as part of a save game file.

tale.util — Generic utilities

Utility stuff

class tale.util.**Context**(*driver: Any*, *clock: tale.util.GameDateTime*, *config: Any*, *player_connection: Any*)
A new instance of this context is passed to every command function and obj.destroy. Note that the player object isn't in here because it is already explicitly passed to these functions.

classmethod from_global(*player_connection=None*) → tale.util.Context
Create a Context based on the current global mud_context Should only be used to (re)create a ctx where one is required, and you don't have a ctx argument provided already.

class tale.util.**GameDateTime**(*date_time: datetime.datetime*, *times_realtime: float = 1*)
The datetime class that tracks game time. times_realtime means how much faster the game time is running than real time. The internal 'clock' tracks the time in game-time (not real-time).

add_gametime(*timedelta: datetime.timedelta*) → None
advance the game clock by a time delta expressed in game time

add_realtime(*timedelta: datetime.timedelta*) → None
advance the game clock by a time delta expressed in real time

minus_realtime(*timedelta: datetime.timedelta*) → datetime.datetime
return the game clock minus a time delta expressed in real time

plus_realtime(*timedelta: datetime.timedelta*) → datetime.datetime
return the game clock plus a time delta expressed in real time

sub_gametime(*timedelta: datetime.timedelta*) → None
rewind the game clock by a time delta expressed in game time

sub_realtime(*timedelta: datetime.timedelta*) → None
rewind the game clock by a time delta expressed in real time

class tale.util.**MoneyFormatter**
Display and parsing of money. Supports 'fantasy' and 'modern' style money.

parse(*words: Sequence[str]*) → float
Convert a parsed sequence of words to the amount of money it represents (float)

class tale.util.**MoneyFormatterFantasy**

display(*amount: float*, *short: bool = False*, *zero_msg: str = 'nothing'*) → str
Display amount of money in gold/silver/copper units, base unit=1 gold, 10 silver=1 gold, 10 copper=1 silver

to_float(*coins: Union[str, Dict[str, float]]*) → float
Either a dictionary containing the values per coin type, or a string '11g/22s/33c' is converted to float.

class tale.util.**MoneyFormatterModern**

display(*amount: float*, *short: bool = False*, *zero_msg: str = 'nothing'*) → str
Display amount of money in modern currency (dollars/cents).

to_float(*coins: Union[str, Dict[str, float]]*) → float
Either a dictionary containing the values per coin type, or a string '$1234.55' is converted to float.

tale.util.**authorized**(**privileges*) → Callable
Decorator for callables that need a privilege check. The callable should have an 'actor' argument that is passed an appropriate actor object with .privileges to check against. If they don't match with the privileges given in this decorator, an ActionRefused error is raised.

tale.util.**call_periodically**(*period: float*, *max_period: float = None*)
Decorator to mark a method of a MudObject class to be invoked periodically by the driver. You can set a fixed

period (in real-time seconds) or a period interval in which a random next occurrence is then chosen for every call. Setting the period to 0 or None will stop the periodical calls. The method is called with a 'ctx' keyword argument set to a Context object.

tale.util.**excepthook**(*ex_type*, *ex_value*, *ex_tb*)
An exception hook you can use for `sys.excepthook`, to automatically print detailed tracebacks

tale.util.**format_docstring**(*docstring: str*) → str
Format a docstring according to the algorithm in PEP-257

tale.util.**format_traceback**(*ex_type: Type = None*, *ex_value: Any = None*, *ex_tb: Any = None*, *detailed: bool = True*, *with_self: bool = False*) → List[str]
Formats an exception traceback. If you ask for detailed formatting, the result will contain info on the variables in each stack frame. You don't have to provide the exception info objects, if you omit them, this function will obtain them itself using `sys.exc_info()`.

tale.util.**get_periodicals**(*obj: Any*) → Dict[Callable, Tuple[float, float, float]]
Get the (bound) member functions that are declared periodical via the @call_periodically decorator

tale.util.**parse_duration**(*args: Sequence[str]*) → datetime.timedelta
parses a duration from args like: 1 hour 20 minutes 15 seconds (hour/h, minutes/min/m, seconds/sec/s)

tale.util.**parse_time**(*args: Sequence[str]*) → datetime.time
parses a time from args like: 13:44:59, or like a duration such as 1h 30m 15s

tale.util.**roll_dice**(*number: int = 1*, *sides: int = 6*) → Tuple[int, List[int]]
rolls a number (max 300) of dice with configurable number of sides

tale.util.**sorted_by_name**(*stuff: Iterable[Any]*) → Iterable[Any]
Returns the objects sorted by their name attribute (case insensitive)

tale.util.**sorted_by_title**(*stuff: Iterable[Any]*) → Iterable[Any]
Returns the objects sorted by their title attribute (case insensitive)

tale.util.**storyname_to_filename**(*name: str*) → str
converts the story name to a suitable name for a file on disk

tale.verbdefs — Soul command verbs definitions

A player's 'soul', which provides a lot of possible emotes (verbs).

Written by Irmen de Jong (irmen@razorvine.net) Based on ancient soul.c v1.2 written in LPC by profezzorn@nannymud (Fredrik Hübinette) Only the verb table is more or less intact (with some additions and fixes). The verb parsing and message generation have been rewritten.

The soul parsing has been moved to the Soul class in the base module.

tale.verbdefs.**adjust_available_verbs**(*allowed_verbs: Sequence[str] = None*, *remove_verbs: Sequence[str] = []*, *add_verbs: Dict[str, Tuple] = {}*) → None
Adjust the available verbs

tale.vfs — Virtual File System to load Resources

Virtual file system.

exception tale.vfs.**VfsError**
Something went wrong while using the virtual file system

class tale.vfs.**VirtualFileSystem**(*root_package: str = ''*, *root_path: Union[str, pathlib.Path] = None*, *readonly: bool = True*, *everythingtext: bool = False*)

Simple filesystem abstraction. Loads resource files embedded inside a package directory. If not readonly, you can write data as well. The API is loosely based on a dict. Can be based off an already imported module, or from a file system path somewhere else. If dealing with text files, the encoding is always UTF-8. It supports automatic decompression of .gz, .xz and .bz2 compressed files (as long as they have that extension). It automatically returns the contents of a compressed version of a requested file if the file itself doesn't exist but there is a compressed version of it available.

contents(*path: str = '.'*) → Iterable[str]

Returns the files in the given path. Only works on path based vfs, not for package based vfs.

open_write(*name: str*, *mimetype: str = ''*, *append: bool = False*) → IO[Any]

returns a writable file io stream

validate_path(*path: str*) → str

Validates the given relative path. If the vfs is loading from a package, the path is returned unmodified if it is valid. If the vfs is loading from a file system location, the absolute path is returned if it is valid.

tale.cmds — In-game commands

Package for all mud commands (non-soul)

tale.cmds.**cmd**(*command: str*, **aliases*) → Callable

Decorator to define a parser command function and its verb(s).

tale.cmds.**wizcmd**(*command: str*, **aliases*) → Callable

Decorator to define a 'wizard' command function and verb. It will add a privilege check wrapper. Note that the wizard command (and the aliases) are prefixed by a '!' to make them stand out from normal commands.

tale.cmds.**disable_notify_action**(*func: Callable*) → Callable

decorator to prevent the command being passed to notify_action events

tale.cmds.**disabled_in_gamemode**(*mode: tale.story.GameMode*) → Callable

decorator to disable a command in the given game mode

tale.cmds.**overrides_soul**(*func: Callable*) → Callable

decorator to let the command override (hide) the corresponding soul command

tale.cmds.**no_soul_parse**(*func: Callable*) → Callable

decorator to tell the command processor to skip the soul parse step and just treat the whole input as plain string

tale.cmds.normal — Normal player commands

Normal player commands.

tale.cmds.normal.**do_account**(*player: tale.player.Player*, *parsed: tale.base.ParseResult*, *ctx: tale.util.Context*) → None

Displays your player account data.

tale.cmds.normal.**do_activate**(*player: tale.player.Player*, *parsed: tale.base.ParseResult*, *ctx: tale.util.Context*) → None

Activate something, turn it on, or switch it on.

tale.cmds.normal.**do_brief**(*player: tale.player.Player*, *parsed: tale.base.ParseResult*, *ctx: tale.util.Context*) → None

Configure the verbosity of location descriptions. 'brief' mode means: show short description for locations that you've already visited at least once. 'brief all' means: show short descriptions for all locations even if you've not been there before. 'brief off': disable brief mode, always show long descriptions. 'brief reset': disable brief

mode and forget about the known locations as well. Note that when you explicitly use the 'look' or 'examine' commands, the brief setting is ignored.

tale.cmds.normal.**do_change_email**(*player: tale.player.Player*, *parsed: tale.base.ParseResult*, *ctx: tale.util.Context*) → Generator
Lets you change the email address on file for your account.

tale.cmds.normal.**do_change_pw**(*player: tale.player.Player*, *parsed: tale.base.ParseResult*, *ctx: tale.util.Context*) → Generator
Lets you change your account password.

tale.cmds.normal.**do_cls**(*player: tale.player.Player*, *parsed: tale.base.ParseResult*, *ctx: tale.util.Context*) → None
Clears the screen (if the output device supports it).

tale.cmds.normal.**do_coin**(*player: tale.player.Player*, *parsed: tale.base.ParseResult*, *ctx: tale.util.Context*) → None
Toss a coin.

tale.cmds.normal.**do_combine_many**(*player: tale.player.Player*, *parsed: tale.base.ParseResult*, *ctx: tale.util.Context*) → None
Combine two or more items you are carrying. If successful, this can perhaps result in a new item!

tale.cmds.normal.**do_combine_two**(*player: tale.player.Player*, *parsed: tale.base.ParseResult*, *ctx: tale.util.Context*) → None
Combine two items you are carrying by attaching them, applying them or installing them together. If successful, this can perhaps result in a new item!

tale.cmds.normal.**do_config**(*player: tale.player.Player*, *parsed: tale.base.ParseResult*, *ctx: tale.util.Context*) → None
Show or change player configuration parameters.

tale.cmds.normal.**do_deactivate**(*player: tale.player.Player*, *parsed: tale.base.ParseResult*, *ctx: tale.util.Context*) → None
Deactivate something, turn it of, or switch it off.

tale.cmds.normal.**do_dice**(*player: tale.player.Player*, *parsed: tale.base.ParseResult*, *ctx: tale.util.Context*) → None
Roll a 6-sided die. Use the familiar '3d6' argument style if you want to roll multiple dice.

tale.cmds.normal.**do_drop**(*player: tale.player.Player*, *parsed: tale.base.ParseResult*, *ctx: tale.util.Context*) → Generator
Drop an item (or all items) you are carrying.

tale.cmds.normal.**do_emote**(*player: tale.player.Player*, *parsed: tale.base.ParseResult*, *ctx: tale.util.Context*) → None
Emit a custom 'emote' message literally, such as: 'emote looks stupid.' -> '<player> looks stupid.

tale.cmds.normal.**do_empty**(*player: tale.player.Player*, *parsed: tale.base.ParseResult*, *ctx: tale.util.Context*) → None
Remove the contents from an object.

tale.cmds.normal.**do_examine**(*player: tale.player.Player*, *parsed: tale.base.ParseResult*, *ctx: tale.util.Context*) → None
Examine something or someone thoroughly.

tale.cmds.normal.**do_exits**(*player: tale.player.Player*, *parsed: tale.base.ParseResult*, *ctx: tale.util.Context*) → None
Provides a tiny clue about possible exits from your current location.

tale.cmds.normal.**do_flee**(*player: tale.player.Player*, *parsed: tale.base.ParseResult*, *ctx: tale.util.Context*) → None
Flee/run in a random or given direction, possibly escaping a combat situation, or shaking off pursuers.

tale.cmds.normal.**do_give**(*player: tale.player.Player*, *parsed: tale.base.ParseResult*, *ctx: tale.util.Context*) → Generator
Give something (or all things) you are carrying to someone else.

tale.cmds.normal.**do_help**(*player: tale.player.Player*, *parsed: tale.base.ParseResult*, *ctx: tale.util.Context*) → None
Provides some helpful information about different aspects of the game. Also try 'hint' or 'recap'.

tale.cmds.normal.**do_inventory**(*player: tale.player.Player*, *parsed: tale.base.ParseResult*, *ctx: tale.util.Context*) → None
Show the items you are carrying.

tale.cmds.normal.**do_license**(*player: tale.player.Player*, *parsed: tale.base.ParseResult*, *ctx: tale.util.Context*) → None
Show information about the game and about Tale, and show the software license.

tale.cmds.normal.**do_load**(*player: tale.player.Player*, *parsed: tale.base.ParseResult*, *ctx: tale.util.Context*) → None
Load a previously saved game.

tale.cmds.normal.**do_locate**(*player: tale.player.Player*, *parsed: tale.base.ParseResult*, *ctx: tale.util.Context*) → None
Try to locate a specific item, creature or player.

tale.cmds.normal.**do_look**(*player: tale.player.Player*, *parsed: tale.base.ParseResult*, *ctx: tale.util.Context*) → None
Look around to see where you are and what's around you.

tale.cmds.normal.**do_loot**(*player: tale.player.Player*, *parsed: tale.base.ParseResult*, *ctx: tale.util.Context*) → None
Take all things from something or someone else. Keep in mind that stealing and robbing is frowned upon, to say the least.

tale.cmds.normal.**do_manipulate**(*player: tale.player.Player*, *parsed: tale.base.ParseResult*, *ctx: tale.util.Context*) → None
Manipulate something.

tale.cmds.normal.**do_motd**(*player: tale.player.Player*, *parsed: tale.base.ParseResult*, *ctx: tale.util.Context*) → None
Show the message-of-the-day again.

tale.cmds.normal.**do_open**(*player: tale.player.Player*, *parsed: tale.base.ParseResult*, *ctx: tale.util.Context*) → None
Do something with a door, exit or item, possibly by using something. Example: open door, unlock chest with key

tale.cmds.normal.**do_put**(*player: tale.player.Player*, *parsed: tale.base.ParseResult*, *ctx: tale.util.Context*) → Generator
Put an item (or all items) into something else. If you're not carrying the item, you will first pick it up.

tale.cmds.normal.**do_quit**(*player: tale.player.Player*, *parsed: tale.base.ParseResult*, *ctx: tale.util.Context*) → Generator
Quit the game.

tale.cmds.normal.**do_read**(*player: tale.player.Player*, *parsed: tale.base.ParseResult*, *ctx: tale.util.Context*) → None
Read something.

tale.cmds.normal.**do_save**(*player: tale.player.Player*, *parsed: tale.base.ParseResult*, *ctx: tale.util.Context*) → Generator
Save your game.

tale.cmds.normal.**do_say**(*player: tale.player.Player*, *parsed: tale.base.ParseResult*, *ctx: tale.util.Context*) → None
Say something to people near you.

tale.cmds.normal.**do_show**(*player: tale.player.Player*, *parsed: tale.base.ParseResult*, *ctx: tale.util.Context*) → None
Shows something to someone else.

tale.cmds.normal.**do_stats**(*player: tale.player.Player*, *parsed: tale.base.ParseResult*, *ctx: tale.util.Context*) → None
Prints the gender, race and stats information of yourself, or another creature or player.

tale.cmds.normal.**do_switch**(*player: tale.player.Player*, *parsed: tale.base.ParseResult*, *ctx: tale.util.Context*) → None
Switch something on or off.

tale.cmds.normal.**do_take**(*player: tale.player.Player*, *parsed: tale.base.ParseResult*, *ctx: tale.util.Context*) → None
Take something (or all things) from something or someone else. Keep in mind that stealing and robbing is frowned upon, to say the least.

tale.cmds.normal.**do_tell**(*player: tale.player.Player*, *parsed: tale.base.ParseResult*, *ctx: tale.util.Context*) → None
Pass a message to another player or creature that nobody else can hear. The other player doesn't have to be in the same location as you.

tale.cmds.normal.**do_teststyles**(*player: tale.player.Player*, *parsed: tale.base.ParseResult*, *ctx: tale.util.Context*) → None
Test the text output styling.

tale.cmds.normal.**do_throw**(*player: tale.player.Player*, *parsed: tale.base.ParseResult*, *ctx: tale.util.Context*) → None
Throw something you are carrying at someone or something. If you don't have it yet, you will first pick it up.

tale.cmds.normal.**do_time**(*player: tale.player.Player*, *parsed: tale.base.ParseResult*, *ctx: tale.util.Context*) → None
Query the current date and/or time of day.

tale.cmds.normal.**do_transcript**(*player: tale.player.Player*, *parsed: tale.base.ParseResult*, *ctx: tale.util.Context*) → None
Makes a transcript of your game session to the specified file, or switches transcript off again.

tale.cmds.normal.**do_turn**(*player: tale.player.Player*, *parsed: tale.base.ParseResult*, *ctx: tale.util.Context*) → None
Turn something (rotate it), or turn something on or off.

tale.cmds.normal.**do_use**(*player: tale.player.Player*, *parsed: tale.base.ParseResult*, *ctx: tale.util.Context*) → None
General object use. Most of the time, you'll need to be more specific to say exactly what you want to do with it.

tale.cmds.normal.**do_wait**(*player: tale.player.Player*, *parsed: tale.base.ParseResult*, *ctx: tale.util.Context*) → None
Let someone know you are waiting for them. Alternatively, you can simply Let time pass. For the latter use, you can optionally specify how long you want to wait (in hours, minutes, seconds).

tale.cmds.normal.**do_what**(*player: tale.player.Player*, *parsed: tale.base.ParseResult*, *ctx: tale.util.Context*) → None
Tries to answer your question about what something is. The topics range from game commands to location exits to creature and items. For more general help, try the 'help' command first.

tale.cmds.normal.**do_where**(*player: tale.player.Player*, *parsed: tale.base.ParseResult*, *ctx: tale.util.Context*) → None
Gives some information on your current whereabouts, or that of something else perhaps. Similar to 'locate'.

tale.cmds.normal.**do_who**(*player: tale.player.Player*, *parsed: tale.base.ParseResult*, *ctx: tale.util.Context*) → None
Search for all players, a specific player or creature, and shows some information about them.

tale.cmds.normal.**do_yell**(*player: tale.player.Player*, *parsed: tale.base.ParseResult*, *ctx: tale.util.Context*) → None
Yell something. People in nearby locations will also be able to hear you.

tale.cmds.normal.**take_stuff**(*player: tale.player.Player, items: Iterable[tale.base.Item], container: tale.base.MudObject, where_str: str = ''*) → int
Takes stuff and returns the number of items taken

tale.cmds.wizard — Wizard commands

Wizard commands.

tale.cmds.wizard.**do_accounts**(*player: tale.player.Player*, *parsed: tale.base.ParseResult*, *ctx: tale.util.Context*) → None
Show all registered player accounts

tale.cmds.wizard.**do_add_priv**(*player: tale.player.Player*, *parsed: tale.base.ParseResult*, *ctx: tale.util.Context*) → None
Usage: add_priv <account> <privilege>. Adds a privilege to a user account. It will become active on next login.

tale.cmds.wizard.**do_ban_unban_player**(*player: tale.player.Player*, *parsed: tale.base.ParseResult*, *ctx: tale.util.Context*) → None
Bans/unbans a player from logging into the game.

tale.cmds.wizard.**do_clean**(*player: tale.player.Player*, *parsed: tale.base.ParseResult*, *ctx: tale.util.Context*) → Generator
Destroys all objects contained in something or someones inventory, or the current location (.)

tale.cmds.wizard.**do_clone**(*player: tale.player.Player*, *parsed: tale.base.ParseResult*, *ctx: tale.util.Context*) → Generator
Clone an item or living directly from the room or inventory, or from an object in the module path

tale.cmds.wizard.**do_clone_vnum**(*player: tale.player.Player*, *parsed: tale.base.ParseResult*, *ctx: tale.util.Context*) → None
Clone an existing item or monster with the given vnum.

tale.cmds.wizard.**do_debug**(*player: tale.player.Player*, *parsed: tale.base.ParseResult*, *ctx: tale.util.Context*) → None
Dumps the internal attribute values of a location (.), item or creature.

tale.cmds.wizard.**do_destroy**(*player: tale.player.Player*, *parsed: tale.base.ParseResult*, *ctx: tale.util.Context*) → Generator
Destroys an object or creature.

tale.cmds.wizard.**do_events**(*player: tale.player.Player*, *parsed: tale.base.ParseResult*, *ctx: tale.util.Context*) → None
Dump pending actions.

tale.cmds.wizard.**do_force**(*player: tale.player.Player*, *parsed: tale.base.ParseResult*, *ctx: tale.util.Context*) → None
Force another living being into performing a given command.

tale.cmds.wizard.**do_go_vnum**(*player: tale.player.Player*, *parsed: tale.base.ParseResult*, *ctx: tale.util.Context*) → None
Teleport to a specific location or creature, given by its vnum.

tale.cmds.wizard.**do_ls**(*player: tale.player.Player*, *parsed: tale.base.ParseResult*, *ctx: tale.util.Context*) → None
List the contents of a module path under the library tree (try !ls .items.basic) or in the story's zone module (try !ls zones)

tale.cmds.wizard.**do_move**(*player: tale.player.Player*, *parsed: tale.base.ParseResult*, *ctx: tale.util.Context*) → None
Move something or someone to another location (.), item or creature. This may work around possible restrictions that could prevent stuff to be moved around normally. For instance you could use it to pick up items that are normally fixed in place (move item to playername).

tale.cmds.wizard.**do_pdb**(*player: tale.player.Player*, *parsed: tale.base.ParseResult*, *ctx: tale.util.Context*) → None
Starts a Python debugging session. (Only available in IF mode)

tale.cmds.wizard.**do_pubsub**(*player: tale.player.Player*, *parsed: tale.base.ParseResult*, *ctx: tale.util.Context*) → None
Give an overview of the pubsub topics.

tale.cmds.wizard.**do_reload**(*player: tale.player.Player*, *parsed: tale.base.ParseResult*, *ctx: tale.util.Context*) → None
Reload the given python module under the library tree (try !reload .items.basic) or one of the story's zone module (try !reload zones.town). This is not always reliable and may produce weird results just like when reloading modules that are still used in python!

tale.cmds.wizard.**do_remove_priv**(*player: tale.player.Player*, *parsed: tale.base.ParseResult*, *ctx: tale.util.Context*) → None
Usage: remove_priv <account> <privilege>. Remove a privilege from a user account. If the account is currently logged in, it will be forced to log off.

tale.cmds.wizard.**do_return**(*player: tale.player.Player*, *parsed: tale.base.ParseResult*, *ctx: tale.util.Context*) → None
Return a player to the location where they were before a teleport.

tale.cmds.wizard.**do_server**(*player: tale.player.Player*, *parsed: tale.base.ParseResult*, *ctx: tale.util.Context*) → None
Dump some server information.

tale.cmds.wizard.**do_set**(*player: tale.player.Player*, *parsed: tale.base.ParseResult*, *ctx: tale.util.Context*) → None
Set an internal attribute of a location (.), object or creature to a new value. Usage is: set xxx.fieldname=value (you can use Python literals only)

tale.cmds.wizard.**do_show_vnum**(*player: tale.player.Player*, *parsed: tale.base.ParseResult*, *ctx: tale.util.Context*) → None
Show the vnum of a location (.) or an object/living, or when you provide a vnum as arg, show the object(s) with that vnum. Special arguments: items/livings/locations/exits to show the known vnums of that class of objects.

tale.cmds.wizard.**do_teleport**(*player: tale.player.Player*, *parsed: tale.base.ParseResult*, *ctx: tale.util.Context*) → None
Teleport to a location or creature, or teleport a creature to you. '!teleport .module.path.to.creature' teleports that creature to your location. '!teleport_to .module.path.to.object' teleports you to that location or creature's location. '!teleport_to zones.zonename.locationname' teleports you to the given location in a zone from the story. '!teleport playername' teleports that player to your location. '!teleport_to playername' teleports you to the location of that player. '!teleport_to @start' teleports you to the starting location for wizards.

tale.cmds.wizard.**do_wiretap**(*player: tale.player.Player*, *parsed: tale.base.ParseResult*, *ctx: tale.util.Context*) → None
Adds a wiretap to something to overhear the messages they receive. 'wiretap .' taps the room, 'wiretap name' taps a creature with that name, 'wiretap -clear' gets rid of all taps.

tale.cmds.wizard.**lookup_module_path**(*path: str*) → module
Gives the module loaded at the given path such as '.items.basic' or 'zones.town.houses'

tale.cmds.wizard.**teleport_someone_to_player**(*who: tale.base.Living*, *player: tale.player.Player*) → None
helper function for teleport command, to teleport someone to the player

tale.cmds.wizard.**teleport_to**(*player: tale.player.Player*, *location: tale.base.Location*) → None
helper function for teleport command, to teleport the player somewhere

tale.tio.iobase — Base classes for I/O

Basic Input/Output stuff not tied to a specific I/O implementation.

class tale.tio.iobase.**IoAdapterBase**(*player_connection*)
I/O adapter base class

abort_all_input(*player*) → None
abort any blocking input, if at all possible

break_pressed() → None
do something when the player types ctrl-C (break)

clear_screen() → None
Clear the screen

critical_error(*message: str = 'A critical error occurred! See below and/or in the error log.'*) → None
called when the driver encountered a critical error and the session needs to shut down

destroy() → None
Called when the I/O adapter is shut down

output(**lines*) → None
Write some text to the screen. Needs to take care of style tags that are embedded. Implement specific behavior in subclass (but don't forget to call base method)

output_no_newline(*text: str*) → None
Like output, but just writes a single line, without end-of-line. Implement specific behavior in subclass (but don't forget to call base method)

pause(*unpause: bool = False*) → None
pause/ unpause the input loop

render_output(*paragraphs: Sequence[Tuple[str, bool]], **params*) → str
Render (format) the given paragraphs to a text representation. It doesn't output anything to the screen yet; it just returns the text string. Any style-tags are still embedded in the text. This console-implementation expects 2 extra parameters: "indent" and "width".

singleplayer_mainloop(*player_connection*) → None
Main event loop for this I/O adapter for single player mode

smartquotes(*text: str*) → str
If enabled, apply 'smart quotes' to the text; replaces quotes and dashes by nicer looking symbols

write_input_prompt() → None
write the input prompt '>>'

tale.tio.iobase.**strip_text_styles**(*text: Sequence[str]*) → Sequence[str]
remove any special text styling tags from the text (you can pass a single string, and also a list of strings)

tale.tio.console_io — Text-console I/O

Console-based input/output.

class tale.tio.console_io.**ConsoleIo**(*player_connection: tale.player.PlayerConnection*)
I/O adapter for the text-console (standard input/standard output).

abort_all_input (*player: tale.player.Player*) → None
abort any blocking input, if at all possible

break_pressed() → None
do something when the player types ctrl-C (break)

clear_screen() → None
Clear the screen

install_tab_completion (*driver: tale.driver.Driver*) → None
Install tab completion using readline, or prompt_toolkit, if available

output (**lines*) → None
Write some text to the screen. Takes care of style tags that are embedded.

output_no_newline (*text: str*) → None
Like output, but just writes a single line, without end-of-line.

pause (*unpause: bool = False*) → None
pause/ unpause the input loop

render_output (*paragraphs: Sequence[Tuple[str, bool]], **params*) → str
Render (format) the given paragraphs to a text representation. It doesn't output anything to the screen yet; it just returns the text string. Any style-tags are still embedded in the text. This console-implementation expects 2 extra parameters: "indent" and "width".

singleplayer_mainloop (*player_connection: tale.player.PlayerConnection*) → None
Main event loop for the console I/O adapter for single player mode

write_input_prompt() → None
write the input prompt '>>'

tale.tio.tkinter_io — Tkinter GUI I/O

GUI input/output using Tkinter.

class tale.tio.tkinter_io.**TkinterIo** (*config*, *player_connection*)
Tkinter-GUI based Input/Output adapter.

abort_all_input (*player*) → None
abort any blocking input, if at all possible

clear_screen() → None
Clear the screen

critical_error (*message: str = 'A critical error occurred! See below and/or in the error log.'*) → None
called when the driver encountered a critical error and the session needs to shut down

destroy() → None
Called when the I/O adapter is shut down

output (**lines*) → None
Write some text to the screen. Needs to take care of style tags that are embedded.

output_no_newline (*text: str*) → None
Like output, but just writes a single line, without end-of-line.

pause (*unpause: bool = False*) → None
pause/ unpause the input loop

render_output (*paragraphs: Sequence[Tuple[str, bool]], **params*) → str
Render (format) the given paragraphs to a text representation. It doesn't output anything to the screen yet; it just returns the text string. Any style-tags are still embedded in the text. This tkinter-implementation expects no extra parameters.

singleplayer_mainloop (*player_connection*) → None
Main event loop for this I/O adapter for single player mode

tale.tio.if_browser_io — Web browser GUI I/O (single-player)

Webbrowser based I/O for a single player ('if') story.

class tale.tio.if_browser_io.**HttpIo** (*player_connection: tale.player.PlayerConnection, wsgi_server: wsgiref.simple_server.WSGIServer*)
I/O adapter for a http/browser based interface. This doubles as a wsgi app and runs as a web server using wsgiref. This way it is a simple call for the driver, it starts everything that is needed.

clear_screen () → None
Clear the screen

convert_to_html (*line: str*) → str
Convert style tags to html

destroy () → None
Called when the I/O adapter is shut down

output (**lines*) → None
Write some text to the screen. Needs to take care of style tags that are embedded. Implement specific behavior in subclass (but don't forget to call base method)

output_no_newline (*text: str*) → None
Like output, but just writes a single line, without end-of-line. Implement specific behavior in subclass (but don't forget to call base method)

pause (*unpause: bool = False*) → None
pause/ unpause the input loop

render_output (*paragraphs: Sequence[Tuple[str, bool]], **params*) → str
Render (format) the given paragraphs to a text representation. It doesn't output anything to the screen yet; it just returns the text string. Any style-tags are still embedded in the text. This console-implementation expects 2 extra parameters: "indent" and "width".

singleplayer_mainloop (*player_connection: tale.player.PlayerConnection*) → None
mainloop for the web browser interface for single player mode

class tale.tio.if_browser_io.**TaleWsgiApp** (*driver: tale.driver.Driver, player_connection: tale.player.PlayerConnection, use_ssl: bool, ssl_certs: Tuple[str, str, str]*)
The actual wsgi app that the player's browser connects to. Note that it is deliberatly simplistic and ony able to handle a single player connection; it only works for 'if' single-player game mode.

class tale.tio.if_browser_io.**TaleWsgiAppBase** (*driver: tale.driver.Driver*)
Generic wsgi functionality that is not tied to a particular single or multiplayer web server.

wsgi_internal_server_error (*start_response: Callable, message: str = ''*) → Iterable[bytes]
Called when an internal server error occurred

wsgi_internal_server_error_json (*start_response: Callable, message: str = ''*) → Iterable[bytes]
Called when an internal server error occurred, returns json response rather than html

wsgi_invalid_request(*start_response: Callable[..., None]*) → Iterable[bytes]
Called if invalid http method.

wsgi_not_found(*start_response: Callable[..., None]*) → Iterable[bytes]
Called if Url not found.

wsgi_not_modified(*start_response: Callable[..., None]*) → Iterable[bytes]
Called to signal that a resource wasn't modified

wsgi_redirect(*start_response: Callable*, *target: str*) → Iterable[bytes]
Called to do a redirect

wsgi_redirect_other(*start_response: Callable*, *target: str*) → Iterable[bytes]
Called to do a redirect see-other

tale.tio.if_browser_io.**WsgiStartResponseType**
alias of typing.Callable

tale.tio.mud_browser_io — Web browser GUI I/O (MUD, multi-user)

Webbrowser based I/O for a multi player ('mud') server.

class tale.tio.mud_browser_io.**MudHttpIo**(*player_connection: tale.player.PlayerConnection*)
I/O adapter for a http/browser based interface.

pause(*unpause: bool = False*) → None
pause/ unpause the input loop

singleplayer_mainloop(*player_connection: tale.player.PlayerConnection*) → None
mainloop for the web browser interface for single player mode

class tale.tio.mud_browser_io.**TaleMudWsgiApp**(*driver: tale.driver.Driver*, *use_ssl: bool*, *ssl_certs: Tuple[str, str, str]*)
The actual wsgi app that the player's browser connects to. This one is capable of dealing with multiple connected clients (multi-player).

tale.tio.styleaware_wrapper — Text wrapping

Textwrapper that doesn't count the length of the embedded formatting tags.

class tale.tio.styleaware_wrapper.**StyleTagsAwareTextWrapper**(*width=70*, *initial_indent=''*, *subsequent_indent=''*, *expand_tabs=True*, *replace_whitespace=True*, *fix_sentence_endings=False*, *break_long_words=True*, *drop_whitespace=True*, *break_on_hyphens=True*, *tabsize=8*, ***, *max_lines=None*, *placeholder=' [...]'*)
A TextWrapper subclass that doesn't count the length of Tale's style tags when filling up the lines (the style tags don't have visible width). Unfortunately the line filling loop is embedded in a larger method, that we need to override fully (_wrap_chunks)...

tale.items.bank — Bank definitions (ATM, credit card)

Banks.

class tale.items.bank.**Bank**(*name: str*, *title: str = ''*, ***, *descr: str = ''*, *short_descr: str = ''*)

allow_item_move(*actor: Optional[tale.base.Living]*, *verb: str = 'move'*) → None
Does the item allow to be moved (picked up, given away) by someone? (yes; no ActionRefused is raised)

handle_verb(*parsed: tale.base.ParseResult*, *actor: tale.base.Living*) → bool
Handle a custom verb (specified in the verbs dict). Return True if handled, False if not handled.

init() → None
Secondary initialization/customization. Invoked after all required initialization has been done. You can easily override this in a subclass. It is not needed to call the Item super class init().

load() → None
Load persisted bank account data from the datafile.

max_num_transactions = 1000
An item (such as ATM or cash card) that you can deposit and withdraw money from. The money is then safe when you log out.

save() → None
Save the bank account data to the data file.

tale.items.basic — Item definitions

A couple of basic items that go beyond the few base types.

class tale.items.basic.**Boxlike**(*name: str*, *title: str = ''*, ***, *descr: str = ''*, *short_descr: str = ''*)
Container base class/prototype. The container can be opened/closed. Only if it is open you can put stuff in it or take stuff out of it. You can set a couple of txt attributes that change the visual aspect of this object.

init() → None
Secondary initialization/customization. Invoked after all required initialization has been done. You can easily override this in a subclass. It is not needed to call the Item super class init().

class tale.items.basic.**Drink**(*name: str*, *title: str = ''*, ***, *descr: str = ''*, *short_descr: str = ''*)

class drinkeffects(*drunkness*, *fullness*, *thirst*)

drunkness
Alias for field number 0

fullness
Alias for field number 1

thirst
Alias for field number 2

init() → None
Secondary initialization/customization. Invoked after all required initialization has been done. You can easily override this in a subclass. It is not needed to call the Item super class init().

class tale.items.basic.**Food**(*name: str*, *title: str = ''*, ***, *descr: str = ''*, *short_descr: str = ''*)

init() → None
Secondary initialization/customization. Invoked after all required initialization has been done. You can easily override this in a subclass. It is not needed to call the Item super class init().

class tale.items.basic.**GameClock**(*name: str*, *title: str = ''*, *, *descr: str = ''*, *short_descr: str = ''*)
A clock that is able to tell you the in-game time.

init() → None
Secondary initialization/customization. Invoked after all required initialization has been done. You can easily override this in a subclass. It is not needed to call the Item super class init().

class tale.items.basic.**Light**(*name: str*, *title: str = ''*, *, *descr: str = ''*, *short_descr: str = ''*)

init() → None
Secondary initialization/customization. Invoked after all required initialization has been done. You can easily override this in a subclass. It is not needed to call the Item super class init().

class tale.items.basic.**MagicItem**(*name: str*, *title: str = ''*, *, *descr: str = ''*, *short_descr: str = ''*)

init() → None
Secondary initialization/customization. Invoked after all required initialization has been done. You can easily override this in a subclass. It is not needed to call the Item super class init().

class tale.items.basic.**Money**(*name: str*, *value: float*, *, *title: str = ''*, *short_descr: str = ''*)
Some money that is lying around. When picked up, it's added to the money the creature is carrying.

notify_moved(*source_container: Union[Location, Container, Living]*, *target_container: Union[Location, Container, Living]*, *actor: Optional[tale.base.Living]*) → None
Called when the item has been moved from one place to another

class tale.items.basic.**Note**(*name: str*, *title: str = ''*, *, *descr: str = ''*, *short_descr: str = ''*)
A (paper) note with or without something written on it. You can read it.

init() → None
Secondary initialization/customization. Invoked after all required initialization has been done. You can easily override this in a subclass. It is not needed to call the Item super class init().

class tale.items.basic.**Potion**(*name: str*, *title: str = ''*, *, *descr: str = ''*, *short_descr: str = ''*)

init() → None
Secondary initialization/customization. Invoked after all required initialization has been done. You can easily override this in a subclass. It is not needed to call the Item super class init().

class tale.items.basic.**Scroll**(*name: str*, *title: str = ''*, *, *descr: str = ''*, *short_descr: str = ''*)

init() → None
Secondary initialization/customization. Invoked after all required initialization has been done. You can easily override this in a subclass. It is not needed to call the Item super class init().

class tale.items.basic.**Trash**(*name: str*, *title: str = ''*, *, *descr: str = ''*, *short_descr: str = ''*)
Trash – junked by cleaners, not bought by any shopkeeper.

class tale.items.basic.**Boat**(*name: str*, *title: str = ''*, *, *descr: str = ''*, *short_descr: str = ''*)

init() → None
Secondary initialization/customization. Invoked after all required initialization has been done. You can easily override this in a subclass. It is not needed to call the Item super class init().

class tale.items.basic.**Wearable**(*name: str*, *title: str = ''*, ***, *descr: str = ''*, *short_descr: str = ''*)

init() → None
Secondary initialization/customization. Invoked after all required initialization has been done. You can easily override this in a subclass. It is not needed to call the Item super class init().

class tale.items.basic.**Fountain**(*name: str*, *title: str = ''*, ***, *descr: str = ''*, *short_descr: str = ''*)

init() → None
Secondary initialization/customization. Invoked after all required initialization has been done. You can easily override this in a subclass. It is not needed to call the Item super class init().

tale.items.board — Bulletin board

Bulletin boards.

class tale.items.board.**BulletinBoard**(*name: str*, *title: str = ''*, ***, *descr: str = ''*, *short_descr: str = ''*)
A bulletin board that stores messages. You can read, post, and remove messages, and reply to them.

handle_verb(*parsed: tale.base.ParseResult*, *actor: tale.base.Living*) → bool
Handle a custom verb (specified in the verbs dict). Return True if handled, False if not handled.

init() → None
Secondary initialization/customization. Invoked after all required initialization has been done. You can easily override this in a subclass. It is not needed to call the Item super class init().

load() → None
Load persisted messages from the datafile. Note: only the posts are loaded from the datafile, not the descriptive texts

save() → None
save the messages to persistent data file

Python Module Index

t

Index

E

F

G

H

I

J

K

L

M

N

O

P

R

S

T

U

V

W