Introduction

In this section we'll look at how to get started writing a game with adventurelib.

Starting a project

The first thing you'll need to do is import the good stuff from adventurelib::

```
from adventurelib import *
```

Then, at the bottom of the file, call the `start()` function, which begins the game::

```
start()
```

Save the file.

Because we haven't added any behaviours, this game won't do very much, but we should run it at this point as a "sanity check" that everything is installed. If you're using IDLE, the game should just run, or you can run it at a command prompt using the `python` or `python3` binary.

.. code-block:: bash

```
python3 my_game.py
```

You should be able to get results like the following::

```
> go north
I don't understand 'go north'.

> help
Here is a list of the commands you can give:
?
help
quit
```

Pressing Ctrl+D will quit the game, or you can type the built-in `quit` command.

Adding a command

All of the rest of your code should go in between the `from adventurelib import *` and the `start()` lines.

We can use the :doc: `@when <commands>` syntax to create a command that player can type in order to interact with your game. Let's add a `brush teeth` command::

```
@when("brush teeth")
def brush_teeth():
    print("You brush your teeth. They feel clean.")
```

If you start the game again you can try out the new command::

```
> brush teeth
You brush your teeth. They feel clean.
```

.. _say:

Using long text

Writing rich, descriptive text is your main tool for getting a player to feel immersed in your game.

While Python's built-in `print()` function is useful for displaying output to a user, it is a bit unwieldy when you want to write several lines of text at once. You could write your descriptions like this, using `+` to glue together individual strings::

```
@when("brush teeth")
def brush_teeth():
    print(
        "You squirt a bit too much toothpaste onto your " +
        "brush and dozily jiggle it round your mouth."
    )
```

This can be inconvenient and harder to make changes to. Adventurelib provides a convenience function called `say()` that you can use instead to show longer strings of text to the player. It's intended to be used with triple-quoted strings like this::

```
@when("brush teeth")
def brush_teeth():
    say("""
        You squirt a bit too much toothpaste onto your
        brush and dozily jiggle it round your mouth.
    """)
```

This will clean up the spacing of the string, then wrap the output to the width of the player's screen.

.. code-block:: none

```
> brush teeth
You squirt a bit too much toothpaste onto
your brush and dozily jiggle it round
your mouth.
```

It also supports multiple paragraphs of text, separated by blank lines::

```
@when("brush teeth")
def brush_teeth():
    say("""
        You squirt a bit too much toothpaste onto your
        brush and dozily jiggle it round your mouth.

        Your teeth feel clean and shiny now, as you
        run your tongue over them.
    """)
```

.. code-block:: none

```
> brush teeth
You squirt a bit too much toothpaste onto
your brush and dozily jiggle it round
your mouth.
```

```
Your teeth feel clean and shiny now, as
you run your tongue over them.
```

You do not have to use `say()` over `print()`:

- `print()` will preserve the formatting of the strings you give it. This is sometimes needed; for example, to show a pre-formatted poem, or to display `ASCII art`_.
- Use `say()` to make it easier to output prose, in a way that will be easier for the player to read.

.. _ `ASCII art`: https://en.wikipedia.org/wiki/ASCII_art

Be creative

That's more or less all there is to it. Now you need to think up a good story for your game.

Adventurelib can help with:

- :doc: `Calling your code in response to player commands <commands>`
- :doc: `Moving through interconnected locations <rooms>`
- :doc: `Referring to items and characters by name <items>`

...but you're going to need to use those features to tell a story that players can interact with and get drawn into. You're going to have to write the Python code that enforces the game's rules and lets you tell that story.

Think about:

- Characters
- Locations
- Emotions
- Detailed descriptions
- Expressive language
- How players will experience your game

Good luck and have fun!

Binding commands

Text adventure games respond to commands entered by the player. Some typical commands might be:

- `north`
- `take wand`
- `give wand to wizard`

adventurelib lets the programmer write code that will run when a command is entered. This means you can decide what happens in response to a command. Your code might decide whether the wizard wants the wand and print what the wizard says when he gets it.

Note that you also needs to check that the wizard is here and you have the wand to give to him!

@when decorator

The `@when` decorator is written on the line above a function. The function will then be called when a player types a matching command.

This code will be called when the player types "scream"::

```
@when("scream")
def scream():
    print("You unleash a piercing shriek that reverberates around you.")
```

Note that this is case-insensitive, so it will also be called when the player types "SCREAM" or "sCrEAM"::

```
> scream
You unleash a piercing shriek that reverberates around you.

> SCREAM
You unleash a piercing shriek that reverberates around you.
```

You can put multiple words into the command. You can also write more than one `@when` line, which means the function will be called if any of the commands match. This can make it easier for the player to work out what to type::

```
@when("shout loudly")
@when("shout")
@when("yell")
def yell():
    print("You bellow at the top of your lungs.")
```

And then in game::

```
> yell
You bellow at the top of your lungs.

> shout loudly
You bellow at the top of your lungs.
```

As with the case of what a player types, so too the spacing of what a player types doesn't matter::

```
>   shout        loudly
You bellow at the top of your lungs.
```

All the words you want the player to type have to be in lower case letters.

Capturing values

While you could write separate functions for "take wand" and "take hat", it's more normal to write a single function that will be called when the player types "take *anything*".

This code will be called when the player types "take *anything*", and the words that match the *anything* will be passed into the function so that you can react to what it was they tried to take::

```
@when("take THING")
def take(thing):
    print(f"You take the {thing}.")
```

So, in a game::

```
> take hat
You take the hat.

> take horse
You take the horse.

> take cheeseburger
You take the cheeseburger.
```

Of course, this isn't a very useful function, because it does not check that there is a thing to take! You will have to write the code that does these checks.

Here's another example, where we capture two words::

```
@when("give ITEM to RECIPIENT")
def give(item, recipient):
    print(f"You give the {item} to the {recipient}.")
```

Here are the rules for what you can write:

- All the words you want the player to type have to be in lower case letters.
- Words that you write in CAPITAL LETTERS will match any word the player types.
- For each word you write in CAPITAL LETTERS, the function has to take a parameter with the same name in lowercase letters.
- The function will be called with the names the player typed - but they will be converted to lower case.

Capturing multi-word names

An UPPERCASE name can match multiple words. If your code contains the above example::

```
@when("give ITEM to RECIPIENT")
```

Then a player can type:

.. code-block:: none

```
> give poison apple to evil godmother
```

And your code will receive the values::

```
item = "poison apple"
recipient = "evil godmother"
```

As long as you require players to type some command words between `ITEM` and `RECIPIENT` (`to` in this case), this will do what you expect. But beware of providing a shorter alias::

```
@when("give ITEM RECIPIENT")
```

Adventurelib uses what's called a **greedy algorithm** - "greedy", because the first group will hungrily "eat" as many words as it can. If a player typed:

.. code-block:: none

```
> give poison apple evil godmother
```

Then `ITEM` will "eat" the first three words, and your code will receive the values::

```
item = "poison apple evil"
recipient = "godmother"
```

Which is probably not what you expect!

However, each CAPITALISED word will match at least one word. So `give apple godmother` will do what you expect. Therefore one solution is to make sure every object in the game can be referred to by a single-word name like `apple`. This can work well in simple games, but the drawback is that you would struggle to create puzzles that involve multiple variations on an object:

.. code-block:: none

```
> inventory
You have:
a red apple
a blue apple

> feed red apple to water nymph
The nymph sticks out her tongue and shivers unenthusiastically.

> feed blue apple to water nymph
The nymph's eyes widen as you take out the blue apple. She dashes
towards you and snatches it from your hands, and then immediately
turns and runs towards the small door.

Glancing back towards you momentarily, she wordlessly tosses you
a slender, silver-blue key, and a moment later is gone.
```

It is probably best to require words like `to`, `with` and `on`, so that adventurelib knows how to split up a phrase::

@when('give ITEM to RECIPIENT')

@when('use ITEM on TARGET')

@when('hit TARGET with WEAPON')

Additional parameters to commands

In some cases, you might like to use a function to handle a number of similar commands.

You can pass additional keyword arguments to the `@when` decorator which will be passed into the handler function whenever that version of the command line matched.

For example::

```
@when('shout', action='bellow')
@when('yell', action='holler')
@when('scream', action='shriek')
def shout(action):
    print(f'You {action} loudly.')
```

Calling @when functions yourself

Even though you've written a `@when` function and it will be called automatically when the player enters that command, you can still call the function yourself normally.

For example, if you write a `look` command, you can call this from other commands, such as when you enter a room:

.. code-block:: python :emphasize-lines: 11

```
@when('look')
def look():
    print(current_room)


@when('go north'):
def go_north():
    global current_room
    current_room = current_room.north
    print('You go north.')
    look()
```

Command Contexts

In some games, a command might only be available in certain contexts, or might change its behaviour in some contexts.

The most simple way of checking if a command can be used right now is to add an `if` statement:

.. code-block:: python

```
@when('exit')
def exit_room():
    global current_room
    if current_room.outside:
        current_room = current_room.outside
    else:
        say("Exit what? You're already outside.")
```

This isn't always the best way. In some cases there are just too many different conditions to check, and you would end up writing too many `if` / `else` statements. This can be useful in situations like these:

- If you have levels then certain actions might only be available in one of the levels.
- If you have a menu - a main menu, or an inventory menu perhaps - then you might have a different set of commands in that menu.
- If you can "unlock" certain commands as you progress through the game.

The **command context** system allows you to configure some of your commands to be available in certain contexts only.

To do this, pass a `context=` keyword argument to the `@when` decorator:

.. code-block:: python :emphasize-lines: 1

```
@when('cast SPELL', context='wonderland')
def cast(spell):
    say(f"You cast the spell.")
```

Now this command will be completely hidden in help and in the game::

```
> cast fireball
I don't understand 'cast fireball'.
```

This command will only become active when we set the context to match. You can set and get the context using `set_context()` and `get_context()`:

.. function:: adventurelib.set_context(new_context)

```
Set the current command context to ``new_context``.

Pass ``None`` to clear the current context.
```

.. function:: adventurelib.get_context()

```
Get the current command context.
```

So for example:

.. code-block:: python

```
@when('enter mirror')
def enter_mirror():
    if get_context() == 'wonderland':
        say('There is no mirror here.')
    else:
        set_context('wonderland')
        say('You step into the silvery surface, which feels wet and cool.')
        say('You realise that clicking your heels will let you return.')


@when('click heels', context='wonderland')
def click_heels(spell):
    set_context(None)
    say('The moment your heels touch the world rearranges around you.')
```

Now you can transition between the different contexts:o

.. code-block:: none

```
> enter mirror
You step into the silvery surface, which feels wet and cool.
You realise that clicking your heels will let you return.

> help
enter mirror
cast SPELL
click heels

> enter mirror
There is no mirror here.

> cast fireball
You cast the spell.

> click heels
The moment your heels touch the world rearranges around you.

> cast fireball
I don't understand 'cast fireball'.
```

```
> click heels
I don't understand 'click heels'.
```

Note that any commands specified without passing `context=` will be available in all contexts.

You might want to call `set_context()` before you call `start()` in order to set the context that the game will start in.

.. tip::

```
Note that if you are not in the right context, the command will not appear
at all. Beware of confusing your users with appearing and disappearing
commands.
```

Context Hierarchies ''''''''''''''''

Contexts may be nested inside other contexts. To do this, use a `.` character to separate different levels of the context hierarchy:

.. code-block:: python

```
@when('land', context='wonderland.flying')
def land():
    set_context('wonderland')
    say('You gradually drop until you feel the earth beneath your feet.')
```

When the current context is `wonderland.flying`, all the `wonderland` commands are available as well as `wonderland.flying` commands and all commands specified without `context=`.

When the current context is `wonderland`, the `land` command will not be available::

```
You dance through the sky like a feather on the wind.

> land
You gradually drop until you feel the earth beneath your feet.
```

```
> land
I don't understand 'land'.
```

The most deeply nested context takes priority. You can use this to pass different parameters to a command in different contexts, or call a different function entirely:

.. code-block:: python

```
@when('north', dir='north')
@when('north', dir='south', context='confused')
def go(dir):
    ...

@when('north', context='confused.really')
def confused_north():
    say('The cauliflowers are in bloom this year.')
```

Rooms

Many adventure games - but not all - have a concept of "rooms". A player can explore rooms with some standard movement commands, perhaps finding interesting items that they can use or characters they can speak to. Note that despite the name, a room doesn't have to be a room of a house. You could use rooms to describe any concept of location, in order to tell your story:

- Drifting in space
- On top of a hill
- Underneath the floorboards
- Nowhere

Adventurelib provides a helper object called `Room`, that can be used within your program. You don't have to use this object in order to create the impression of rooms though. You can do it with creative use of `@when` functions.

Creating a room

Rooms are created by passing a description. Rich descriptions that convey a story to the user are very important to make your text adventure immersive, so try to write at least a couple of sentences. ::

```
from adventurelib import *

space = Room("""
You are drifting in space. It feels very cold.

A slate-blue spaceship sits completely silently to your left,
its airlock open and waiting.
""")

spaceship = Room("""
The bridge if the spaceship is shiny and white, with thousands
of small, red, blinking lights.
""")
```

Next you'll want the ability to move between rooms. adventurelib doesn't track what room the player is in; this is your responsibility!::

```
# current_room will be a global variable. Let's start out in
# space, so assign the 'space' room from above.
current_room = space


@when('enter airlock')
def enter_spaceship():
    # To set a global variable from within a function you have
    # to include the 'global' keyword, to avoid creating a
    # local variable instead.
    global current_room

    # Got to check if this action can be done here
    if current_room is not space:
        print('There is no airlock here.')
        return

    current_room = spaceship

    # You should include some narrative for every action to
    # ensure the transition doesn't feel abrupt.
    print(
```

```
        "You heave yourself into the airlock and slam your " +
        "hand on the button to close the outer door."
    )

    # Show the room description to indicate we have arrived.
    print(current_room)
```

Storing attributes on rooms

Part of the reason for rooms is to have different objects or contexts for the story. Some actions could only be possible in some rooms. You can assign arbitrary attribute names to an object in order to track the state of a room or what actions can be performed there. You can also set attributes on the `Room` object, which apply for all rooms::

```
Room.can_scream = True  # The default for all rooms
space.can_scream = False  # Set a value for a specific room.

@when('scream')
def scream():
    if current_room.can_scream:
        print(
            "You unleash a piercing shriek that " +
            "reverberates around you."
        )
    else:
        print(
            "You try to yell but there's no sound " +
            "in the vacuum of space."
        )
```

If you access an attribute that doesn't exist on a room, an `AttributeError` will be raised, so ensure that you either set an attribute on every single room or set a default value on `Room`.

Directions and exits

Many text adventure games let players explore a system of rooms freely, using common commands such as `north`, `south`, `east` and `west`.

`Room` objects support these compass point directions by default. If you assign a room as the `north` attribute of another room, then you can traverse this relationship. ::

```
space.north = spaceship
```

Then one could access the room to the north of the current room using normal attribute access::

```
current_room.north
```

The key feature of the directions system is that these references are **bi-directional**. adventurelib knows that `north` is the opposite of `south`, so these relationships automatically hold::

```
>>> space.north is spaceship
True
>>> spaceship.south is space
True
```

Exits

Rooms have a couple of methods that allow you to query what exits they have.

These can be useful when writing commands that use the room layout (such as moving or looking in a direction).

.. function:: room.exit(direction)

```
Get the Room that is linked in direction (eg. ``north``). Returns ``None``
if there is no room in that direction.
```

.. function:: room.exits()

```
Get a list of direction names where a direction is set.
```

Moving between rooms

To follow the links you've defined you could define separate `north`, `south`, `east` and `west` handlers - but the code would be mostly the same, and this is annoying to type and make changes to.

Instead, we can define one function and use several different `@when` lines to define the directions we will go. Each one will pass a direction in which to go.::

```
@when('north', direction='north')
@when('south', direction='south')
@when('east', direction='east')
@when('west', direction='west')
def go(direction):
    global current_room
    room = current_room.exit(direction)
    if room:
        current_room = room
        print(f'You go {direction}.')
        look()
```

Then in game::

```
> north
You go north.
There is a polar bear here.

> south
You go south.
It is a bright, sunny day.
```

These can be some of the most heavily used command, so you could also provide alias commands `n`, `s`, `e` and `w` as a convenience::

```
@when('north', direction='north')
@when('south', direction='south')
@when('east', direction='east')
@when('west', direction='west')
@when('n', direction='north')
@when('s', direction='south')
```

```
@when('e', direction='east')
@when('w', direction='west')
def go(direction):
    ...
```

Adding more directions

While `north`, `south`, `east` and `west` are built into adventurelib, you don't have to use them. You can also register new directions, so long as you give an opposite. You would typically do this at the top of the file, before you define any rooms::

```
Room.add_direction('up', 'down')
Room.add_direction('enter', 'exit')

tent = Room(...)
camp = Room(...)
river = Room(...)
camp.enter = tent
camp.down = river
```

Items

Many games will allow players to pick up objects. Also perhaps some actions in the game will cause players to receive objects, such as when given them by a character.

To support this, adventurelib provides two classes that work together: `Item` and `Bag`.

Defining an item

The `Item` class represents an item. The most important feature is that items can be referred to by a number of names. This means that you can use a descriptive name for the item in output that you show to the user, but allow the user to refer to the item by a shorter name. In game, the interaction might be as follows:

.. code-block:: none

```
> look
You are in a dirt-stained and litter-strewn alley behind
the cinema.
There is a broken broom here.

> take broom
You take the broom.

> inventory
You have:
a broken broom
```

To represent an object like this in the game, construct an Item object::

```
broom = Item('a broken broom', 'broom')
```

The first name you give is the default name for the item, which can be inserted into strings::

```
print(f'You sweep away cobwebs with {broom}.')
```

All the other names you give are *aliases* for the object. See :ref: `bags` for how to select items based on what the player types.

Item Attributes ''''''''''''

Items can be assigned arbitrary attributes, which can be used to set properties that your `@when` handlers can use for game logic.

Like :doc: `Room <rooms>` , you can assign class attributes on `Item` in order to have a default that applies for all items that aren't set specifically.

For example::

```
Item.colour = 'grey'

mug = Item('mug')
mug.colour = 'red'

@when('look at ITEM')
def look(item):
```

```
    obj = inventory.find(item)
    if not item:
        print(f"You do not have a {item}.")
    else:
        print(f"It's a sort of {obj.colour}-ish colour.")
```

Definite/indefinite articles ''''''''''''''''''''''''

In English, we refer to objects using indefinite articles "a", "an" and "some" when we're talking about some object out of a class of objects, and "the" when we're talking about a specific one or specific group.

If you're not writing in English, you may have similar grammatical considerations - genders, word endings etc.

You can store these variations on the name as attributes on the item for use in constructing grammatical sentences - here we use `def_name`, but use whatever attributes you like::

```
apples = Item('some apples', 'apples', 'apple')
apples.def_name = 'the apples'

@when('take ITEM')
def take_item(item):
    obj = current_room.items.take(item)
    if not obj:
        print(f'There is no {item} here.')
    else:
        print(f'You take {item.def_name}.')
        inventory.add(obj)
```

Making your sentences obey correct grammar in all case may not be easy - good luck!

.. _bags:

Bags of items

A `Bag` is a collection of items. This does not need to be a literal "bag" that the player is holding - it's a metaphor! You could treat a Room as being bag of items. Or a group of :ref: `characters` could be held in a Bag.

The point of a Bag is to allow you to look up items by the names that players have typed for them. For this purpose, they have these methods:

.. class:: Bag([items])

```
  Construct a bag from a list of items.
```

.. function:: name in bag

```
  Test if the name the player entered is an object in the bag.
```

.. function:: bag.find(name)

```
  Return the item corresponding to a name the player typed, but don't remove
  it from the bag.

  Returns ``None`` if the name didn't match any object in the bag.
```

.. function:: bag.take(name)

```
  Like ``find()``, find the item corresponding to the name the player typed,
  but then remove it from the bag and return it.

  Returns ``None`` if the name didn't match any object in the bag.
```

.. function:: bag.get_random()

```
  Select and return one item from the bag at random, without removing it.
```

.. function:: bag.take_random()

```
  Remove and return one item from the bag at random.
```

But Bags are also sets_ so they **inherit** various methods for modifying and iterating over items in the Bag, most usefully:

.. function:: bag.add(item)

```
Put `item` into the bag if it isn't already in it.
```

.. function:: for item in bag

```
Loop over the items in the bag.
```

.. _sets: https://docs.python.org/3/tutorial/datastructures.html#sets

So, you could model the player's inventory as a Bag::

```
inventory = Bag()

@when('eat ITEM')
def eat(item):
    obj = inventory.take(item)
    if not obj:
        print(f'You do not have a {item}.')
    else:
        print(f'You eat the {obj}.')

@when('inventory')
def show_inventory():
    print('You have:')
    if not inventory:
        print('nothing')
        return
    for item in inventory:
        print(f'* {item}')
```

You could also model the items on the ground in a room as a bag::

```
chapel.items = Bag([
    Item('a golden candlestick', 'candlestick'),
])

@when('take ITEM')
def take(item):
    obj = current_room.items.take(item)
```

```
    if not obj:
        print(f'There is no {item} here.')
    else:
        inventory.add(obj)
        print(f'You take the {obj}.')
```

.. _characters:

Characters

You can treat non-player characters as items also.

You might want to store pronouns for the characters as attributes on the Item object for use in constructing grammatical sentences::

```
wizard = Item('a wizard')
wizard.def_name = 'the wizard'
wizard.subject_pronoun = 'he'
wizard.object_pronoun = 'him'
```

To avoid repeating this for all male and all female characters, consider creating a small subclass (of course, you could do this for any other group of Items that share common attributes)::

```
class MaleCharacter(Item):
    subject_pronoun = 'he'
    object_pronoun = 'him'
```

Then the above example can be written just as::

```
wizard = MaleCharacter('a wizard')
wizard.def_name = 'the wizard'
```

Customisations

Of course, your game will have a unique story, but you can also change adventurelib's behaviour to suit your game.

Below we will discuss some of the possible customisations and why you might want to use them in a game.

Input Prompt

Some games display status information in the prompt, such as health. For example::

```
10HP > attack grue
You flail wildly at the grue, but it neatly side-steps you and
kicks you in the ribs, for 1HP damage.

9HP >
```

'HP' is an abbreviation for 'health points' that comes from classic computer games. But you could use a Unicode heart symbol for this!

Alternatively, you might want to display some status at intervals in the game, unrelated to the actions a player has taken, such as the footsteps in this example::

```
> north
You enter a long, rocky passage dimly lit with flickering torches.
The corridor curves to the east.

You hear footsteps to the east.

> east
There's a small nook here. Sitting on a plinth is a crude idol of
a beat with horns.

You hear footsteps to the north.
>
```

To customise the prompt, write a function that returns what the prompt string should be. Usually it should end with a space. Then assign this function as `adventurelib.prompt` like this::

```
import adventurelib  # Put this at the top of the file

def prompt():
    return '{hp}HP > '.format(hp=player_hp)

adventurelib.prompt = prompt
```

Disabling the help command

In some games, forcing the player to work out what to type is half the fun.

To make this kind of game work, it's important to respond to things that the player types with custom responses, so be prepared to write a lot of `@when` functions that respond to many varieties of input.

However, the built-in `help` / `?` commands would spoil this kind of game by giving all the answers.

You can disable the help by setting `help=False` when calling `start()` ::

```
start(help=False)
```

Customising the "I don't understand" message

When the player types a command that doesn't match any existing `@when` function, adventurelib responds with a basic "I don't understand" message::

```
> jump up and down
I don't understand 'jump up and down'.
```

This could get very boring if users see it a lot!

To customise this, write a function and assign it as `adventurelib.no_command_matches` . This function should accept the input the player typed as its argument, and print any responses::

```
import adventurelib  # Put this at the top of the file

def no_command_matches(command):
    print(random.choice([
        'Huh?',
        'Sorry?',
        'I beg your pardon?'
    ]))

adventurelib.no_command_matches = no_command_matches
```

Credits

adventurelib - easy text adventures