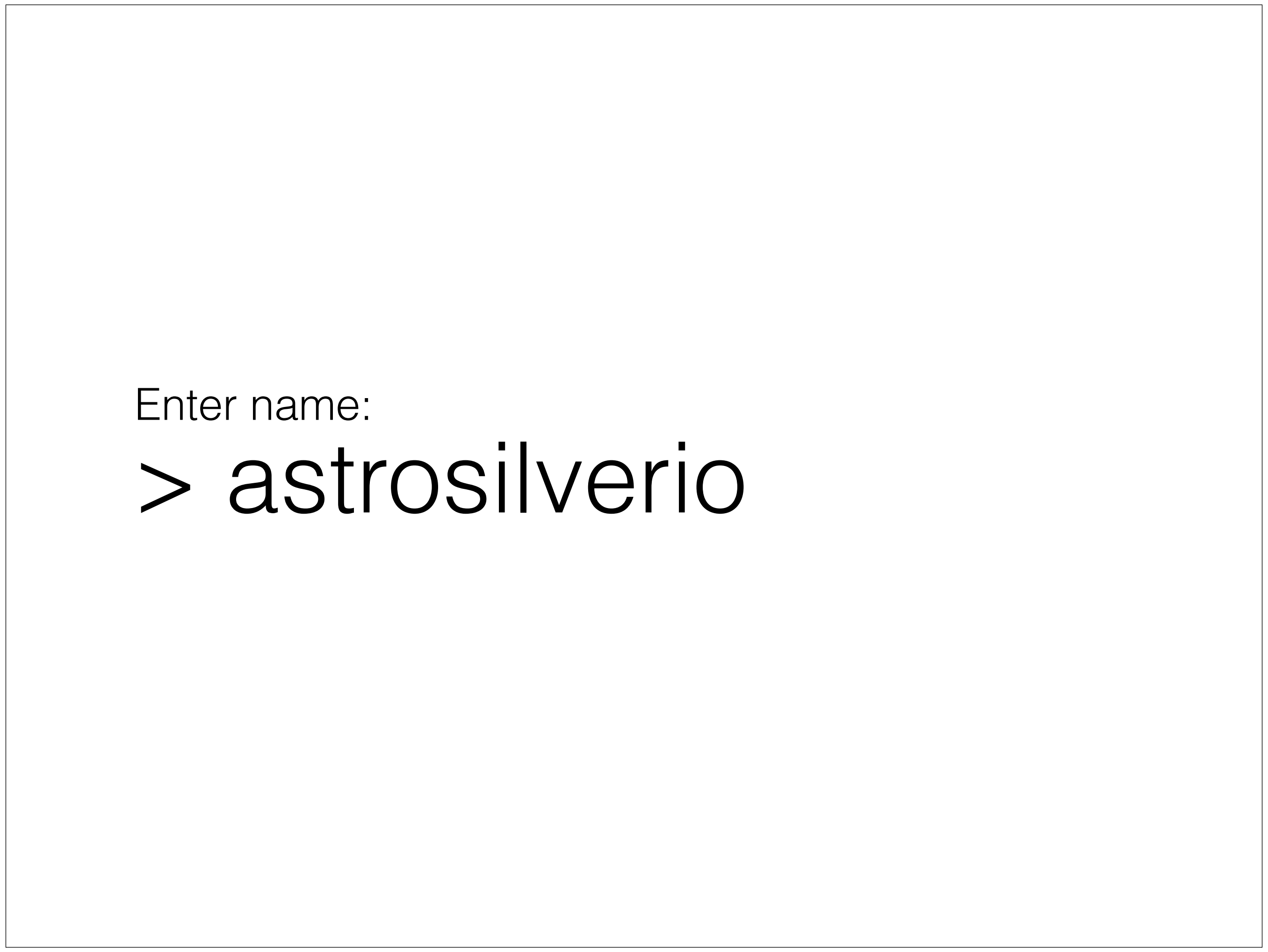
"A huge green fierce snake bars the way";

Or, Building a Text Adventure Game in Python


Enter name:

> astrosilverio

Hi. My name's Katie Silverio, I'm a software engineer at Venmo in NYC, and Recurse Center alum from Fall 2013. I am astrosilverio on all things (github, twitter, and my tech blog on tumblr), and today I'm here to talk a little about a project I've been working on, namely a framework for writing and running text adventure games.<pause>
A text adventure is a game with text-only input and text-only output. There's a slew of classic ones from the 70s and 80s, including Colossal Cave Adventure and Zork. The genre is still going strong, albeit partially under the name of interactive fiction. There's a popular system for creating interactive fiction games called Inform 7, which is remarkable in that Inform 7 code reads almost like English. If you have never played a text adventure, I highly recommend it, particularly HHGGTG.

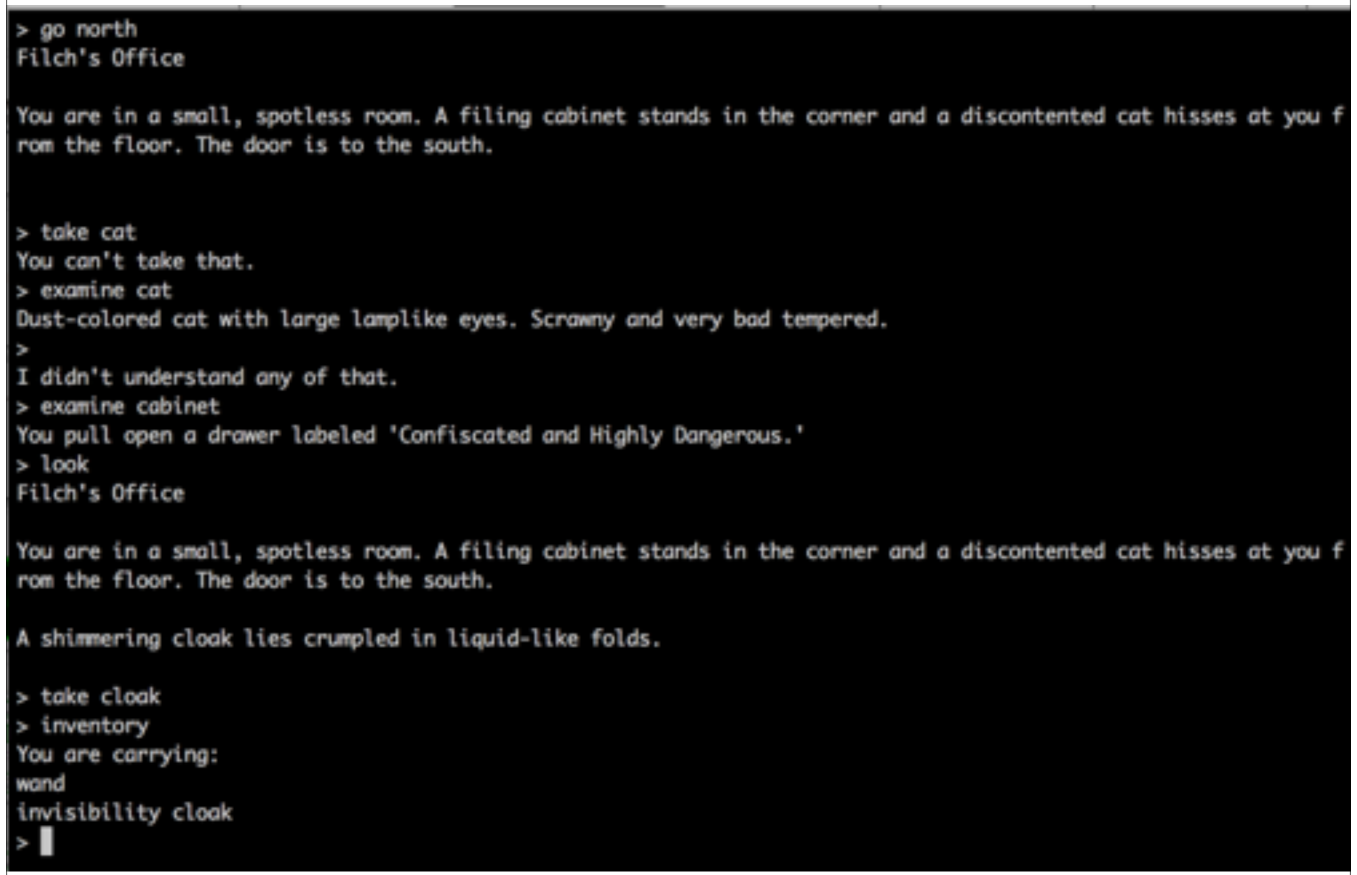
> examine talk

what are we doing here

> How does a text adventure work?

- Why make a text adventure framework?
- How does a text adventure framework work?
- Let's make a framework!

We're here today because a couple of years ago I started writing a Harry Potter themed text adventure game, and I'd like to use that game as an example of how a text adventure game works.

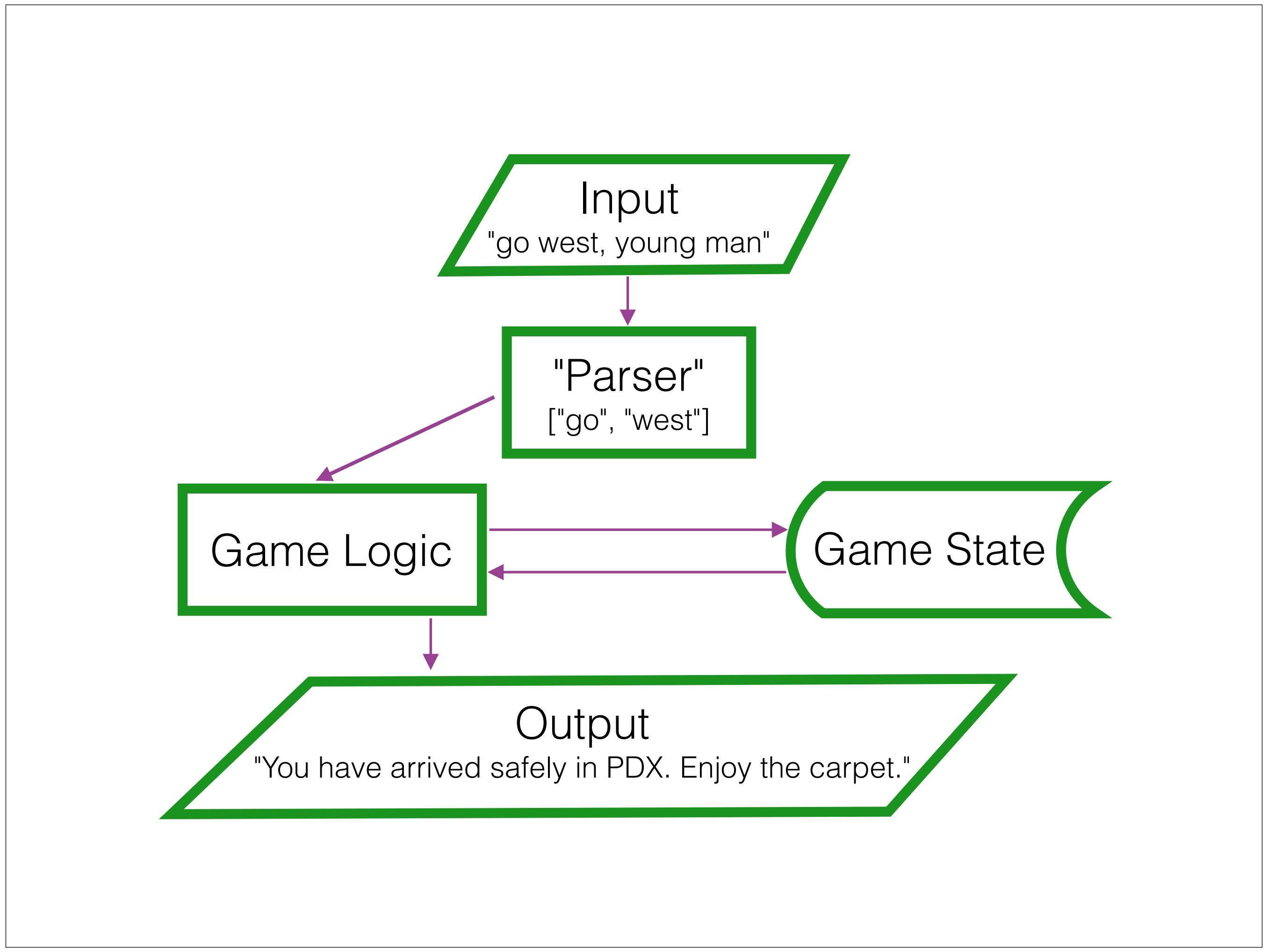


```
> go north
Filch's Office

You are in a small, spotless room. A filing cabinet stands in the corner and a discontented cat hisses at you f
rom the floor. The door is to the south.


> take cat
You can't take that.
> examine cat
Dust-colored cat with large lamplike eyes. Scrawny and very bad tempered.
>
I didn't understand any of that.
> examine cabinet
You pull open a drawer labeled 'Confiscated and Highly Dangerous.'
> look
Filch's Office

You are in a small, spotless room. A filing cabinet stands in the corner and a discontented cat hisses at you f
rom the floor. The door is to the south.

A shimmering cloak lies crumpled in liquid-like folds.

> take cloak
> inventory
You are carrying:
wand
invisibility cloak
> 
```

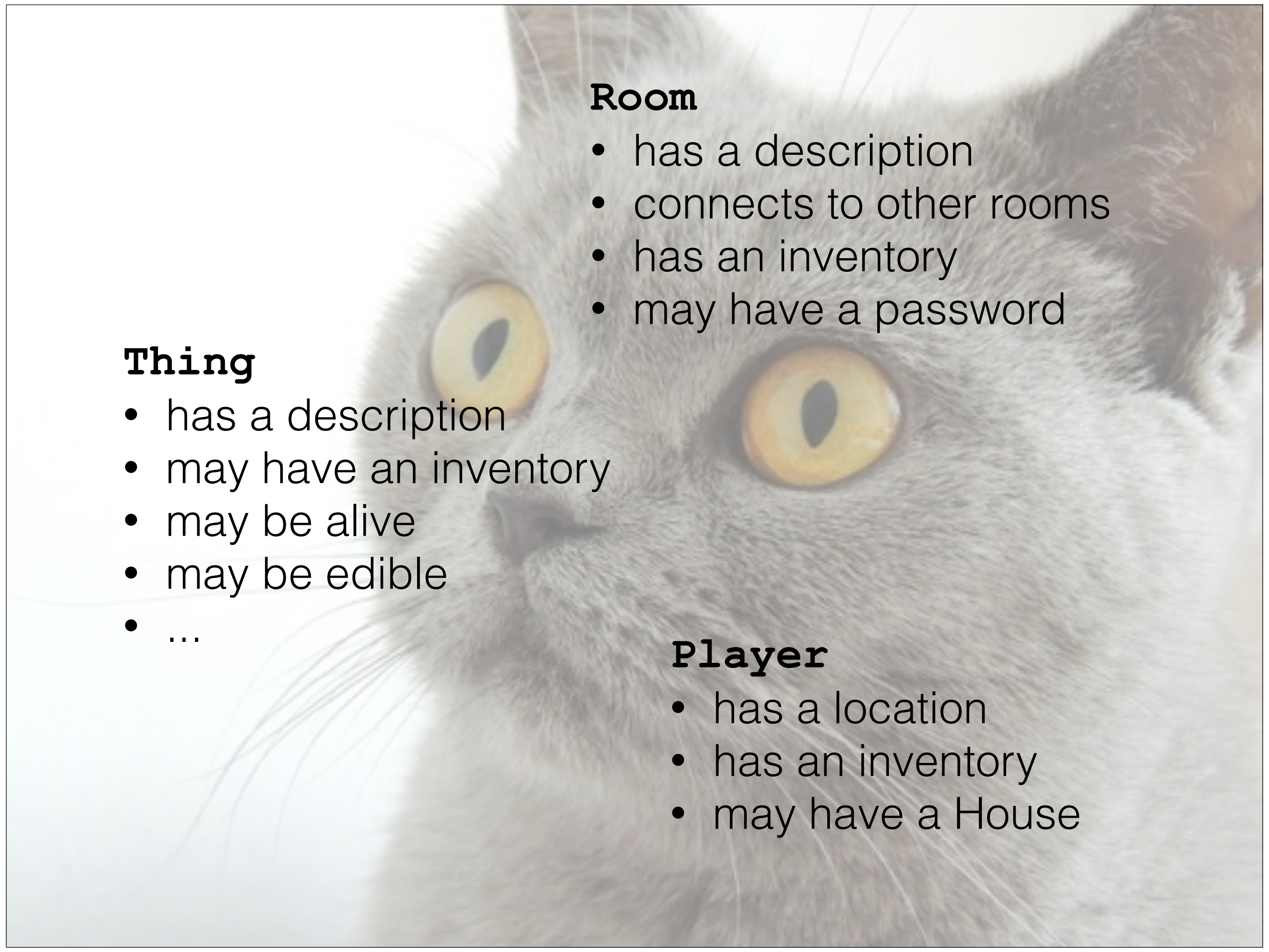
When you play a text adventure there are some things that you'll notice. First off, the game has state. For example, at this point in my Harry Potter game, there is a room. The room contains a cat. The cat is alive. There is a player. The player is in Filch's Office. These states can change -- this player might not always be in this room. The cat might not always be alive. However, there are rules about how that state can change.

If you try and take the cat -- change the cat's state from being in the room to being in the player's possession or inventory -- the game doesn't let you do that. If you try instead to take the invisibility cloak, you can, thereby changing your state and the room's state.<pause>

If you've ever played a Twine game, or another choose-your-own-adventure game, you may already be familiar with this state-machine kind of setup. The difference between a choose-your-own-adventure and a text adventure is that as a player you're not at all limited in what kind of (text) input you can give the game, and the game has a little bit more work cut out for it in interpreting your input.



3:20 Let's condense what we noticed on the last slide into a flow of what the program is doing. When a player user types some input, the game has to figure out what they're talking about -- parse their input into an instruction for the game. Once it thinks it knows what the player wants, it has to try and do that thing. So the game state has a nice conversation with the game rules, and between them they decide whether the change that the player wants to make can happen. Finally, it reports back to the player.



Room

- has a description
- connects to other rooms
- has an inventory
- may have a password

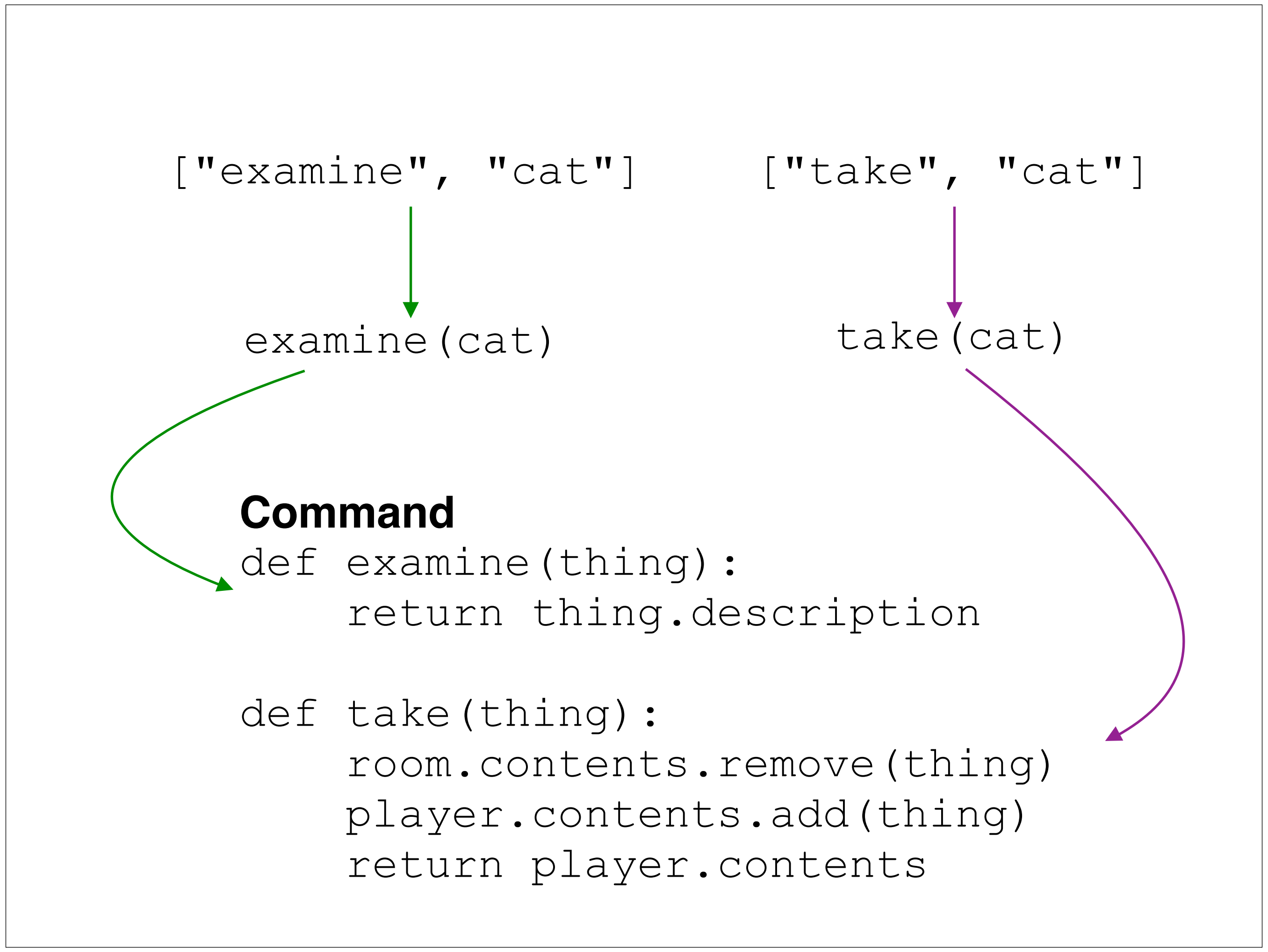
Thing

- has a description
- may have an inventory
- may be alive
- may be edible
- ...

Player

- has a location
- has an inventory
- may have a House

5:00 A brief word about state: For my text adventure game, I chose to implement three types of stateful things. Rooms, like Filch's Office, have descriptions, can connect to other Rooms, and can contain Things and Players (in other words, have an inventory). Things, like the filing cabinet, have descriptions, may have an inventory, may be edible, etc. Players ("you") have a location, an inventory, and if they've progressed far enough in the game to be Sorted, have a House. The entire state of the Hogwarts is represented by the collective state of all the Rooms, Things, and Players in the game.<pause>
Why? The short answer is, it's easier this way. Imagine if the state was NOT split up. The game state at the point where I try to pick up the cat is "Player in Filch's Office, Filch's Office contains cat and cabinet, cabinet contains cloak, player holds wand, player's house is Gryffindor..." and much much more (Hogwarts is a big place with a lot of things in it!). When I try to execute "take cat", the game has to decide if I can go from that initial state (call it State A) to "Player is in Filch's Office, Filch's Office contains cabinet, cabinet contains cloak, player holds wand and cat, player's house is Gryffindor..." etc.<pause>
In theory, the game could do this by looking up in a table of allowed transitions whether it is allowed to go from State A to State B. That is very simple for the game to do. However, generating a complete list of states and allowed transitions between states, is an absolute nightmare for a gamemaker when the number of possible states, and the number of allowable transitions between states, are as huge as they are in even a moderately sized text adventure.

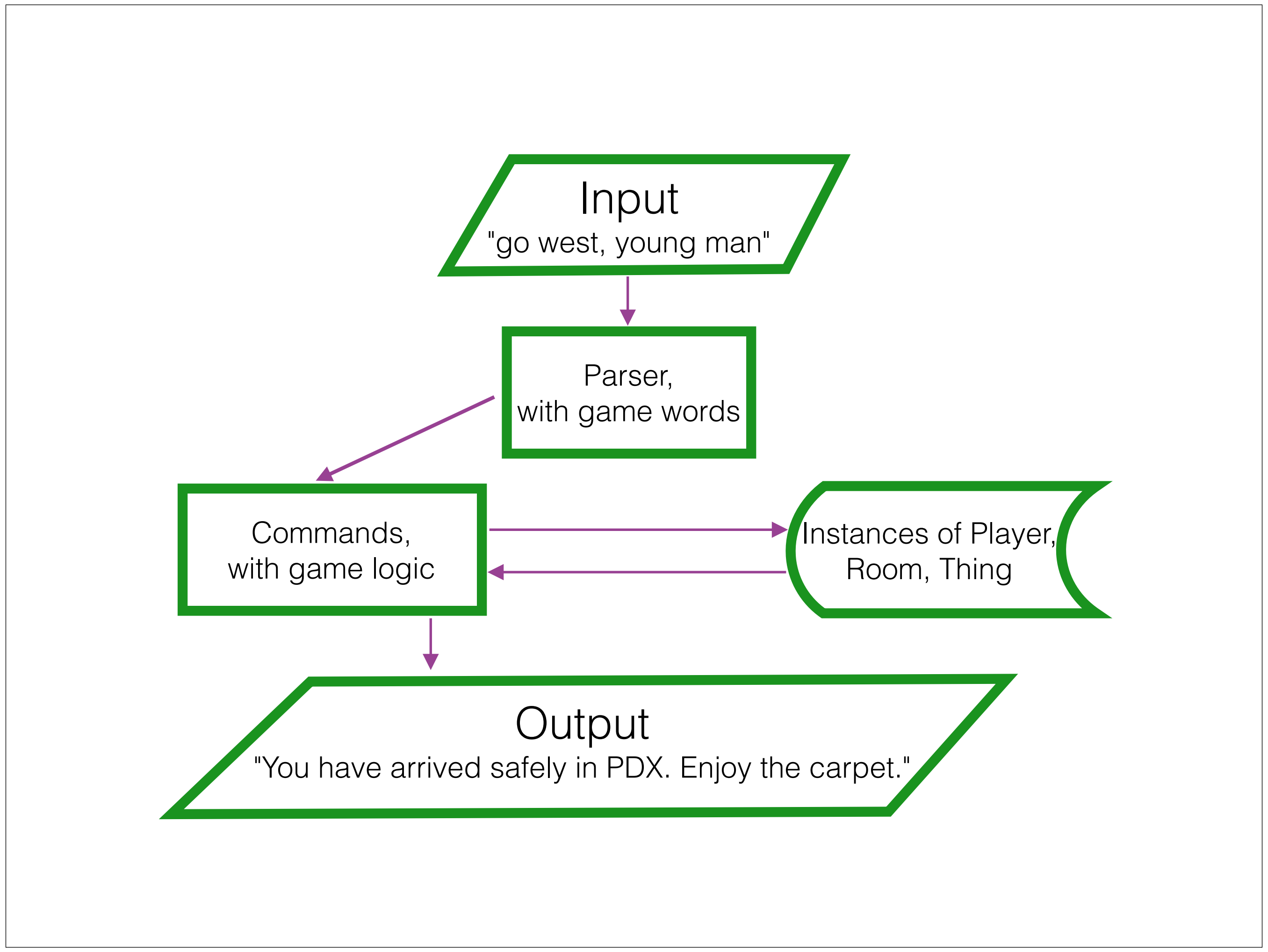


```
["examine", "cat"]      ["take", "cat"]

   examine(cat)           take(cat)
```

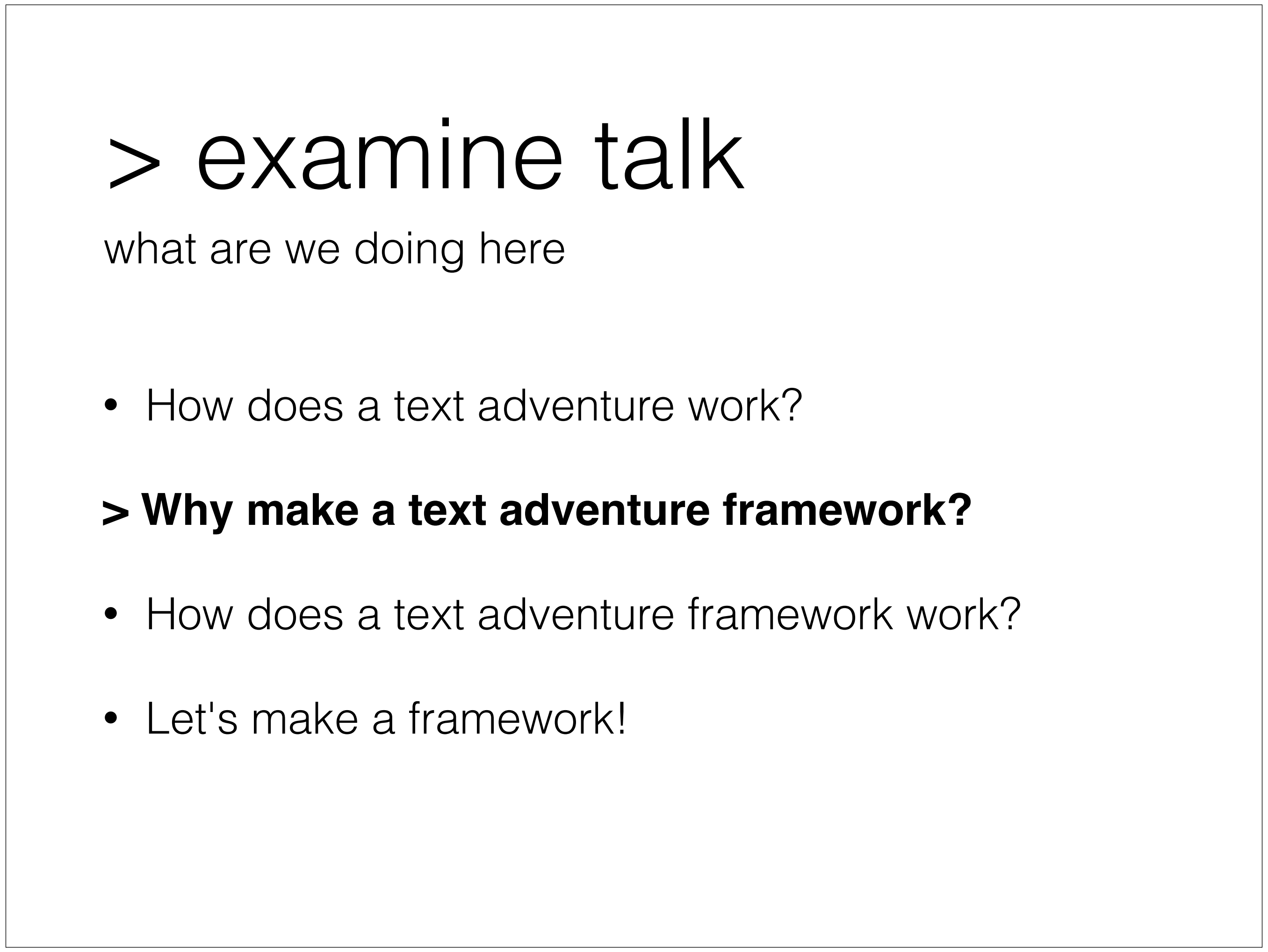
Command

```
def examine(thing):
    return thing.description

def take(thing):
    room.contents.remove(thing)
    player.contents.add(thing)
    return player.contents
```

8:00 So then how does the game decide whether or not a transition is allowable, if it can't look for a link between two global states? Well, in a word: logic. Instead of writing out all the situations in which the player could pick up the cat, the gamemaker instead writes out all the conditions that would have to be true for the player to be able to pick up the cat. I find it easiest to encapsulate known potential player requests as functions that I call Commands, which make the requisite state checks and changes. The logic handler's job on receiving processed input from the parser become calling the correct Command with the appropriate arguments.



11:00 Let's update that flowchart from earlier--the parser talks to the logic handler, which calls the appropriate command (*cough* function), which consults/modifies classes called Player, Room, Thing, which are essentially boxes storing state as attributes.<pause>
And here we have an example of generalization improving UX -- to make it easier for the gamemaker to set and regulate game state, we split the game state up into bits and use commands with logic (if statements) to determine whether the request change can happen. This is the flow that I used for my text adventure game.



> examine talk

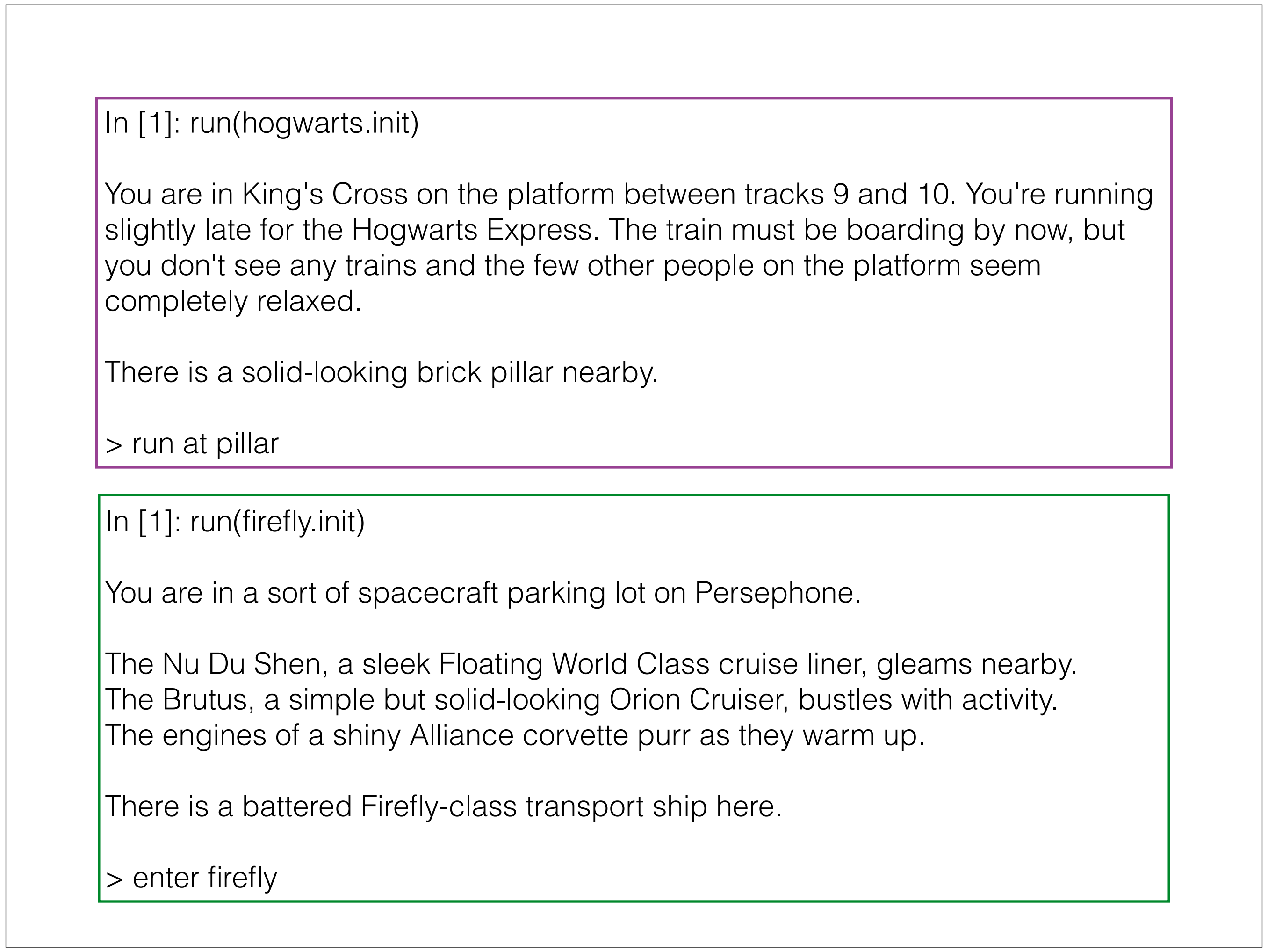
what are we doing here

- How does a text adventure work?

> Why make a text adventure framework?

- How does a text adventure framework work?
- Let's make a framework!

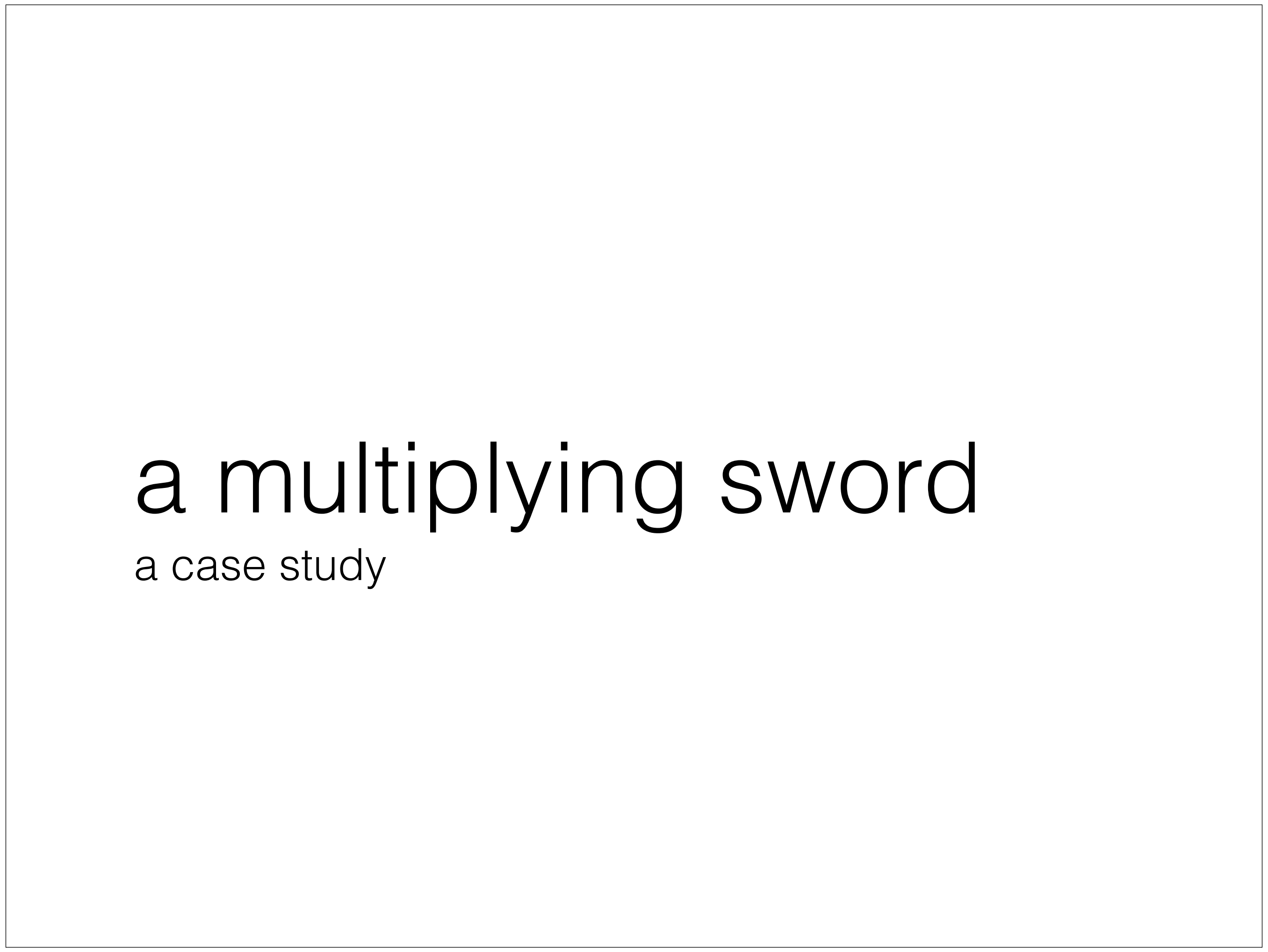
If that works, and it mostly does (bug stories ahead), why bother making a framework?



```
In [1]: run(hogwarts.init)

You are in King's Cross on the platform between tracks 9 and 10. You're running
slightly late for the Hogwarts Express. The train must be boarding by now, but
you don't see any trains and the few other people on the platform seem
completely relaxed.

There is a solid-looking brick pillar nearby.

> run at pillar
```

```
In [1]: run(firefly.init)

You are in a sort of spacecraft parking lot on Persephone.

The Nu Du Shen, a sleek Floating World Class cruise liner, gleams nearby.
The Brutus, a simple but solid-looking Orion Cruiser, bustles with activity.
The engines of a shiny Alliance corvette purr as they warm up.

There is a battered Firefly-class transport ship here.

> enter firefly
```

Well, the short answer is that so I can make more than one text adventure game. As much as I love working on my Harry Potter game, it's not inconceivable that sooner or later I'd want to make another text adventure game. While I could, with my text adventure game code, create a map of new Rooms with new Things and new Players for, say, the Firefly 'verse, I wouldn't be able to reuse any of my logic code, which would try to turn lights on if the player user types "lumos". In order to be able to write more than one text adventure, I need to have a system that not only will allow me to use or not use stateful things like Filch's Office, but also will allow me to use or not use logic such as when to release the Sword of Gryffindor.



a multiplying sword

a case study

I promised some bug stories in the abstract, and you've been very good listening to me go on about state and logic and parsers for a bit, so let's take a detour and talk about the Sword of Gryffindor.

a multiplying sword

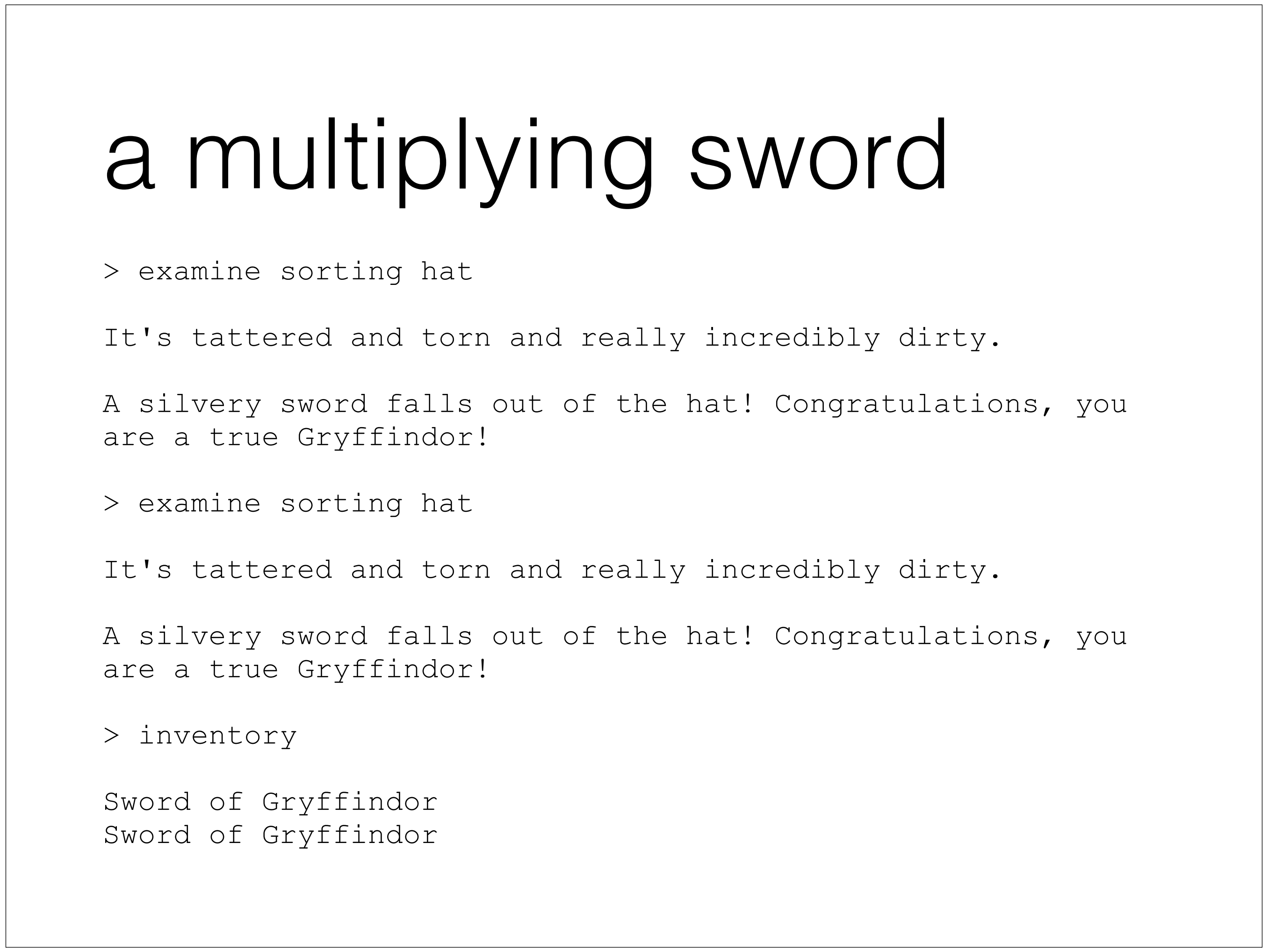
```
> examine sorting hat

It's tattered and torn and really incredibly dirty.
```

13:00 Most text adventures have a command called "examine", which returns a detailed description of, well, the thing you're examining. If you examine, say, the sorting hat, you'd expect a comment describing the hat's appearance. But, in the Harry Potter universe, the sorting hat, when examined by a member of Gryffindor house, will release the Sword of Gryffindor.

a multiplying sword

```
> what's my house

You are in Gryffindor House. You are brave and chivalrous,
and possibly obnoxious. You probably have red hair.

> examine sorting hat

It's tattered and torn and really incredibly dirty.

A silvery sword falls out of the hat! Congratulations, you
are a true Gryffindor!
```

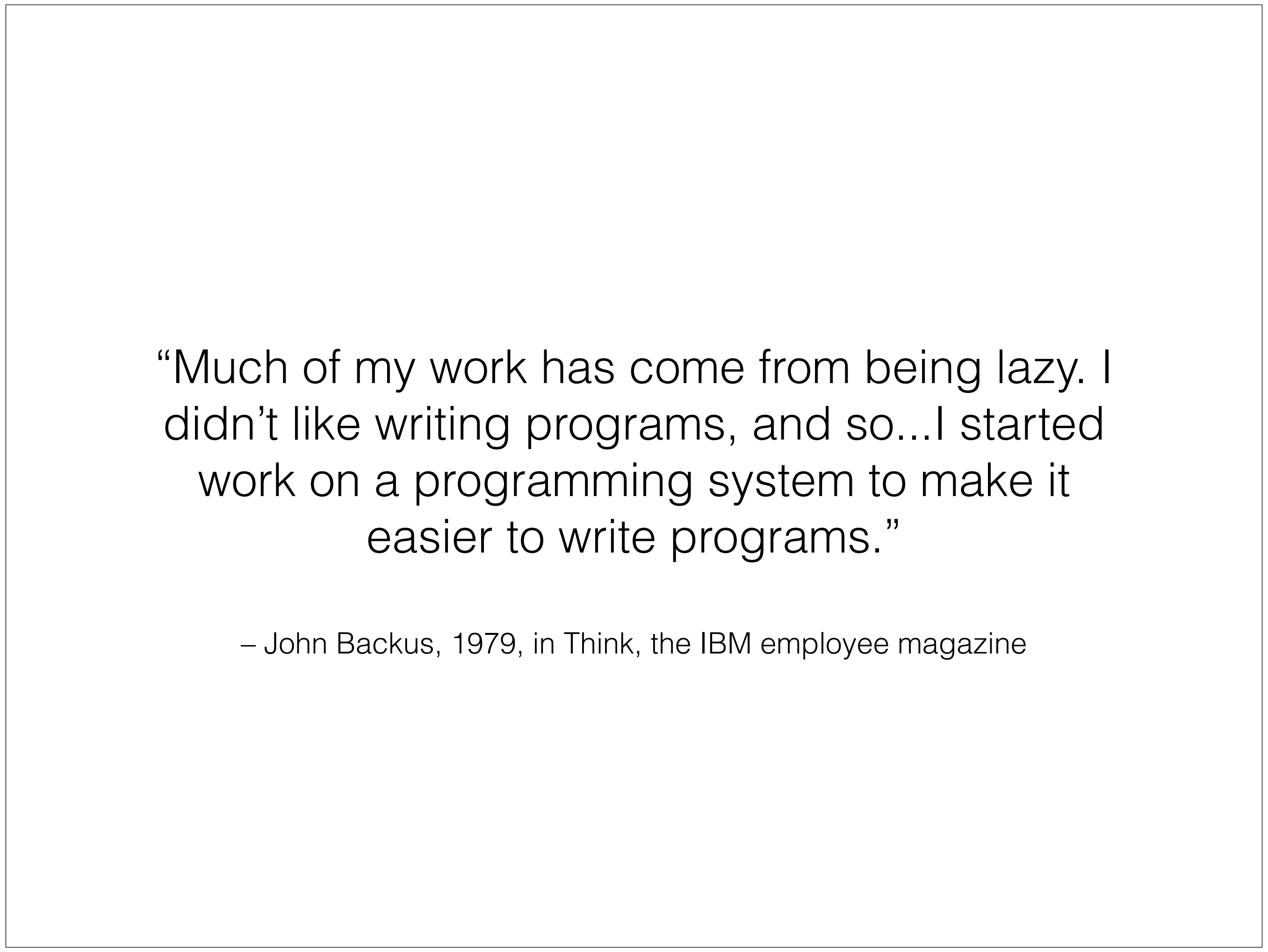
13:35 So I added some logic to the function `examine` that checks if the thing being examined is the sorting hat and if the player is a Gryffindor. If both of those checks are true, than I change the output of examine to include a line about the sword, and as a side effect add the sword to the player's inventory. This implementation results in a lot of complexity being added to an otherwise simple function for the sake of an edge case. There's a surprise state change, and to cap it all off, it doesn't work.



a multiplying sword

```
> examine sorting hat

It's tattered and torn and really incredibly dirty.

A silvery sword falls out of the hat! Congratulations, you
are a true Gryffindor!

> examine sorting hat

It's tattered and torn and really incredibly dirty.

A silvery sword falls out of the hat! Congratulations, you
are a true Gryffindor!

> inventory

Sword of Gryffindor
Sword of Gryffindor
```

14:15 Imagine that your house happens to be Gryffindor, so you can release the sword from the hat. You examine the hat, get the Sword of Gryffindor, examine the hat again, get another the Sword of Gryffindor... Either there's an arbitrary number of Godric Gryffindors in the world, all of whom have swords and are involved with the education of British teenagers, or something's wrong here.<pause>The solution that I hit on was checking to see if the sword existed in the game somewhere, and only dropping the sword from the hat if the sword wasn't already in the game. This is a solution to the problem. But it's a bad solution, partly because complex if-statement-ridden code is dirty code, and partly because the UX for the game developer is terrible. If I want to write another text adventure, my `examine` command has to be completely rewritten. The experience that I, as a gamemaker want, is the freedom to write game-specific state and rules for changing state without having to rewrite code that will be common to all text adventures.


"Much of my work has come from being lazy. I didn't like writing programs, and so...I started work on a programming system to make it easier to write programs."

– John Backus, 1979, in Think, the IBM employee magazine

15:30 In short, if a gamemaker is writing a Harry Potter-themed text adventure game, they should be writing Harry Potter things and not text adventure game things. That's what I mean by a text adventure framework--a program that lets me write other programs.

> examine talk

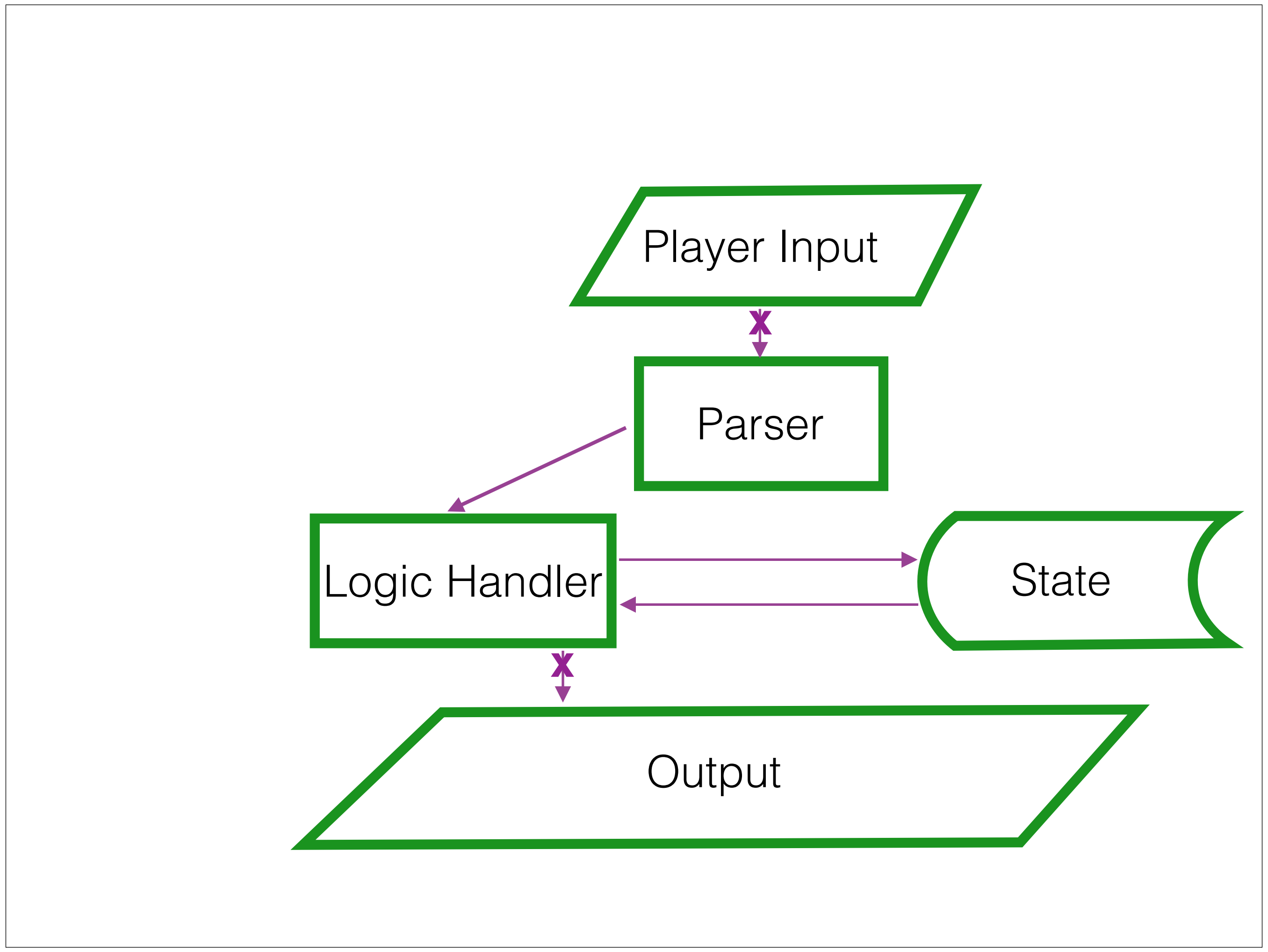
what are we doing here

- How does a text adventure work?
- Why make a text adventure framework?

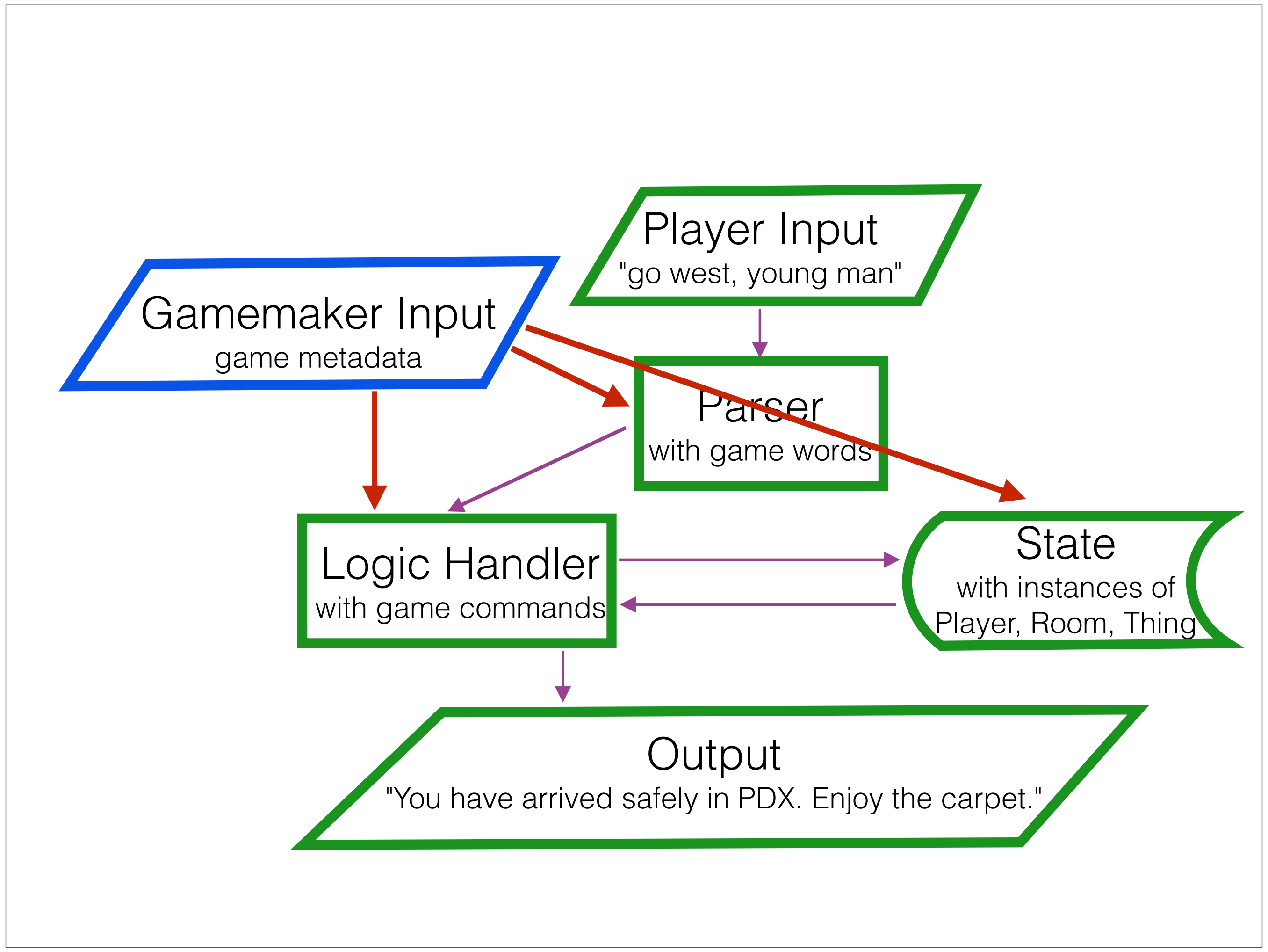
> How does a text adventure framework work?

- Let's make a framework!

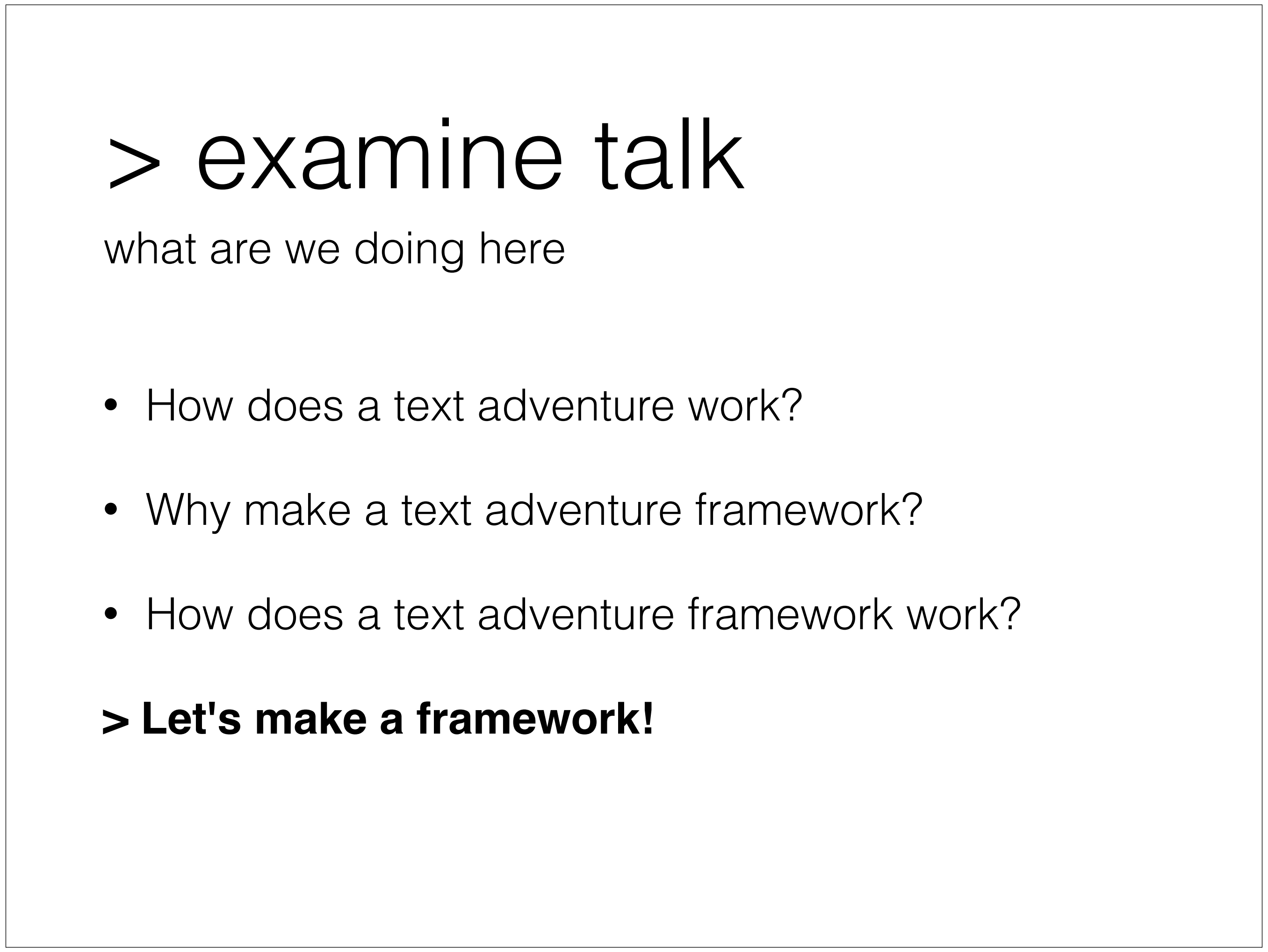
16:00 How is implementing a framework different from implementing a game?



16:30 Short answer: it's not really all that different. From the player's perspective, the UX of playing a game made with a framework is no different from a stand-alone text adventure, which means the same things have to be done with player input as before.<pause>The difference is that the engine--the set of parser, logic handler, and state--has no information about the game, and therefore doesn't function on its own.



To make a functioning game, the engine must be initialized with game-specific data. With a framework, there are two users -- the player user, who types things and expects text responses in return, and the gamemaker user, who writes the game metadata and feeds it to the framework engine to produce a game identity that will handle the input from the player. <pause>



> examine talk

what are we doing here

- How does a text adventure work?
- Why make a text adventure framework?
- How does a text adventure framework work?

> Let's make a framework!

18:00 Now the project becomes abstracting all reliance on game data out of the three major pieces of the engine--parser, logic handler, and state.

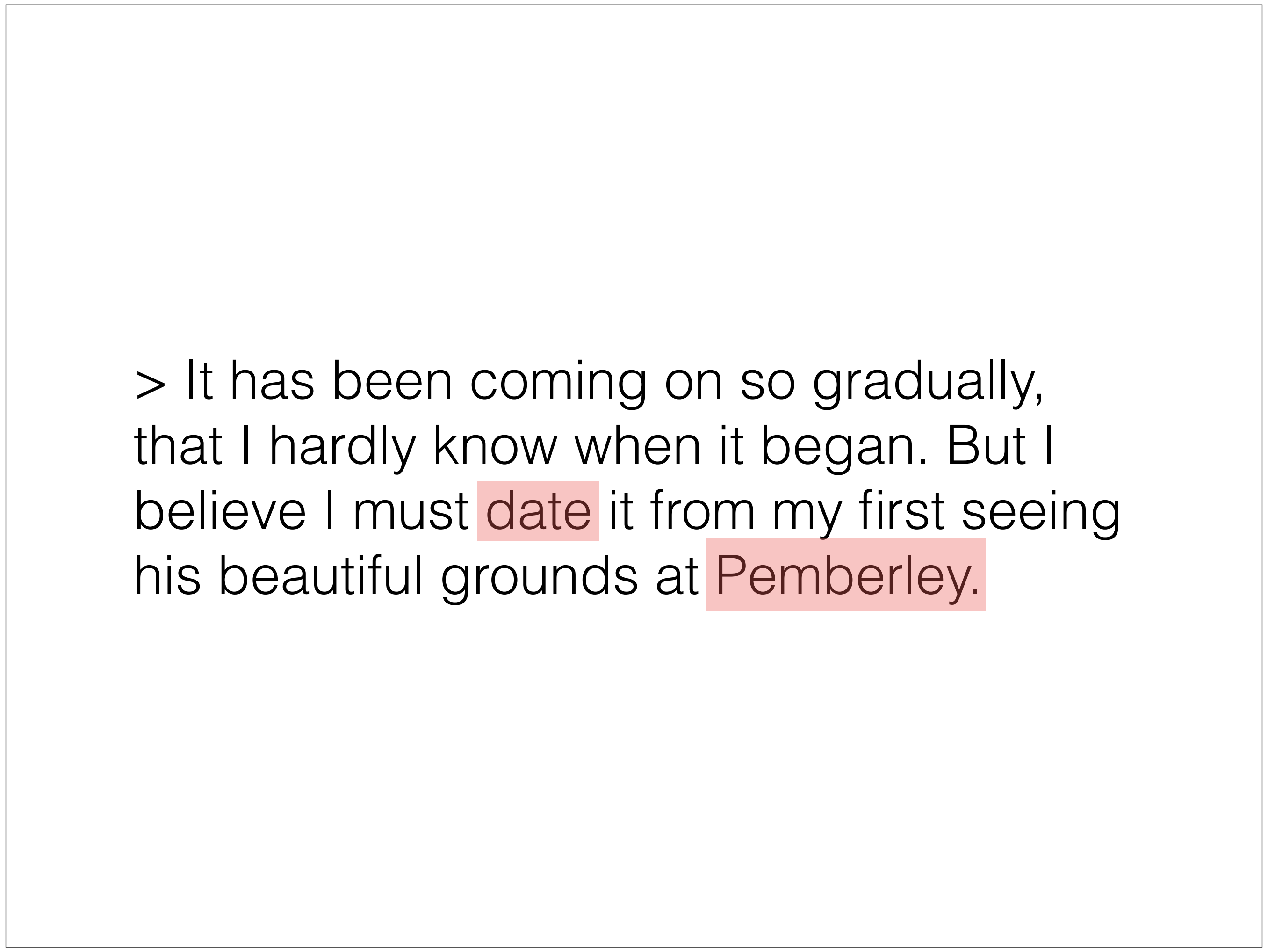
> activate translator

a brief word about abstracting parsing

Let's start with the parser. The parser's job is to pass an instruction to the logic handler from the player's input, so the only game knowledge it needs is what words to recognize. My text adventure game has a whitelist of accepted words that represent commands, players, things, rooms, and directions -- in other "words", only words that represent concrete things or concepts in the game are recognized.

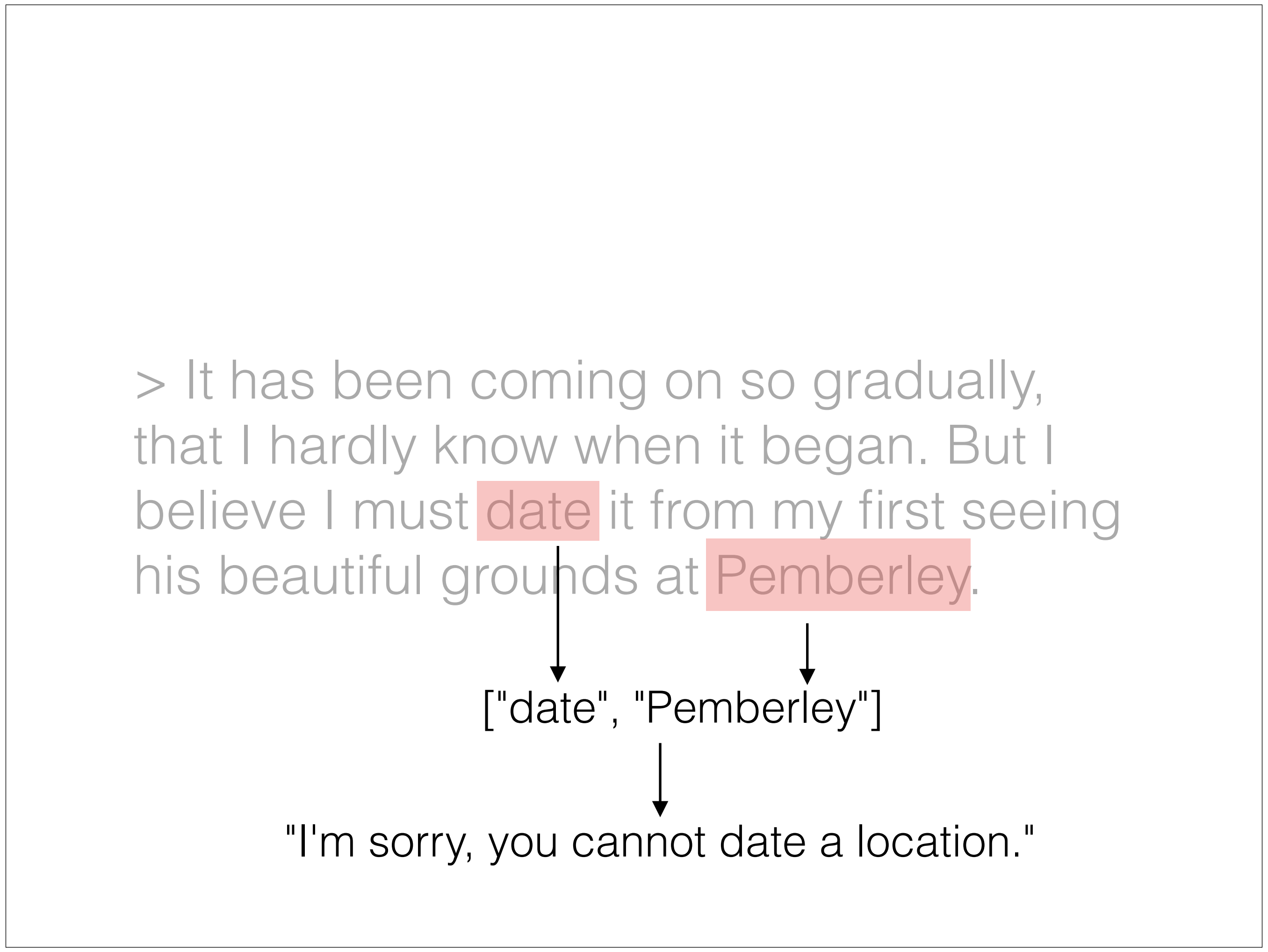
> It has been coming on so gradually, that I hardly know when it began. But I believe I must date it from my first seeing his beautiful grounds at Pemberley.

19:00 Imagine my parser was given this sentence to parse for a Pride and Prejudice themed game.

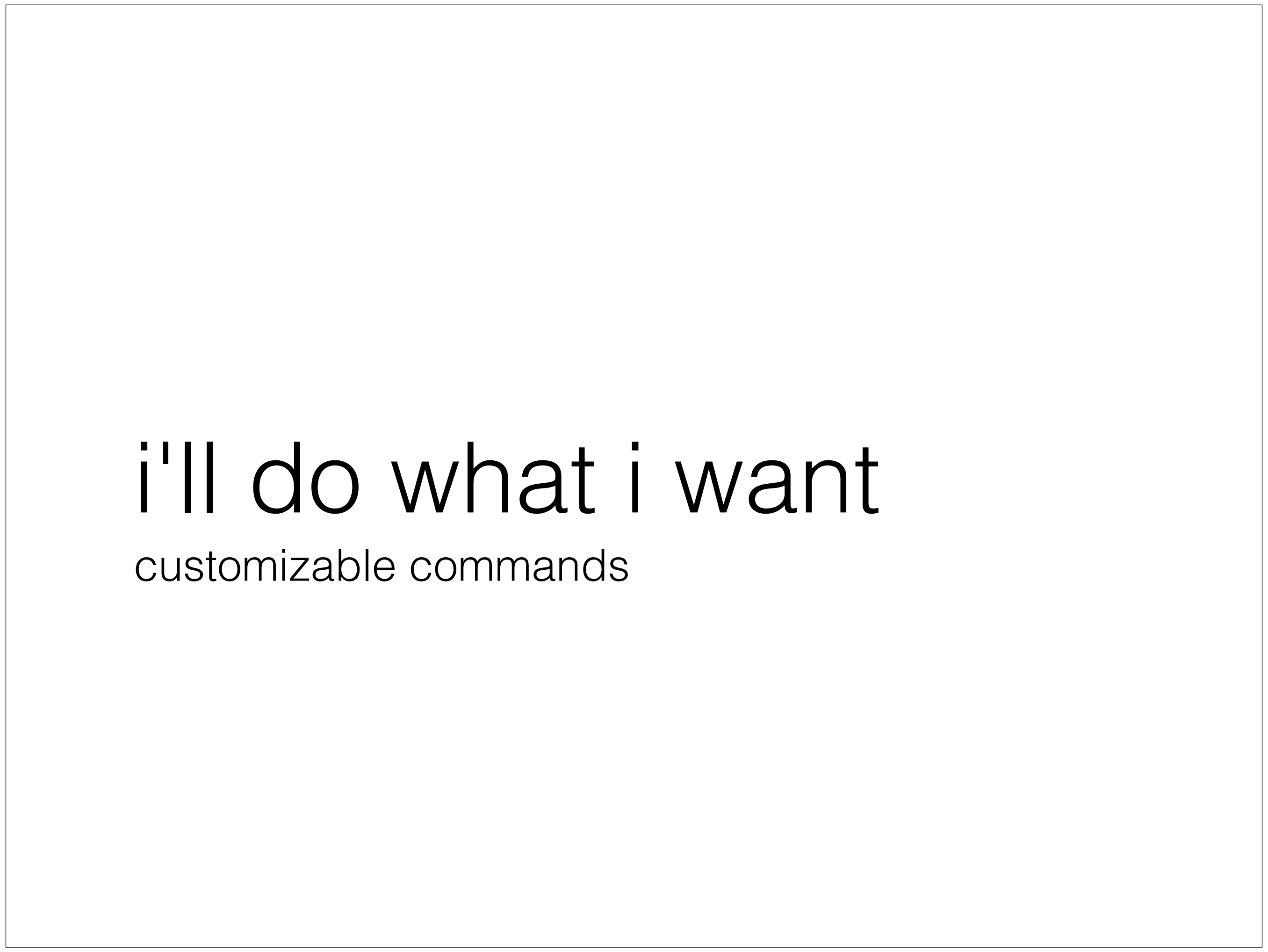


```
> It has been coming on so gradually,
that I hardly know when it began. But I
believe I must date it from my first seeing
his beautiful grounds at Pemberley.
```

Of all the words in this sentence, only two words represent concepts or things in the game: "date", which is a game-specific command that takes a character name as an argument; and "Pemberley", which is a location in the game.


> It has been coming on so gradually, that I hardly know when it began. But I believe I must date it from my first seeing his beautiful grounds at Pemberley.
["date", "Pemberley"]
"I'm sorry, you cannot date a location."

The parser passes ["date", "Pemberley"] off to the logic handler--which doesn't make sense to the logic handler, but hey, the parser tried. This approach means that abstracting game data from the parser becomes a matter of just initializing the parser with a custom list of words.



i'll do what i want

customizable commands

20:20 Abstraction in the logic handler is a little more difficult. We can keep the same general principle--logic handler receives instructions from parser, dispatches instructions to appropriate Command--but Commands have to be totally customizable. I want my HP-themed game to have accio, for example, but I don’t want accio to exist in a great American road-trip adventure. Some commands should come built-in with the engine. Chances are very high that you’ll need go, for example, and you shouldn’t have to re-implement it for every game. But you should be able to modify the built-in commands. Maybe in your road-trip adventure, go west will move you one unit west if you’re not in a car, and will move you and your car 10 units west if you are in a car.

> define command

```
go('wand') => wat
go('south') => gotcha
```

21:30 So how do we do this?
Well, first we have to decide what we mean by a command. What does a command need to do? I expect the command to do something based on my input, if my input is valid, and then return a response. For example, if I type some garbage like go wand, I expect the go command to do nothing and return something like “I don’t know where you want me to go.”

> define command

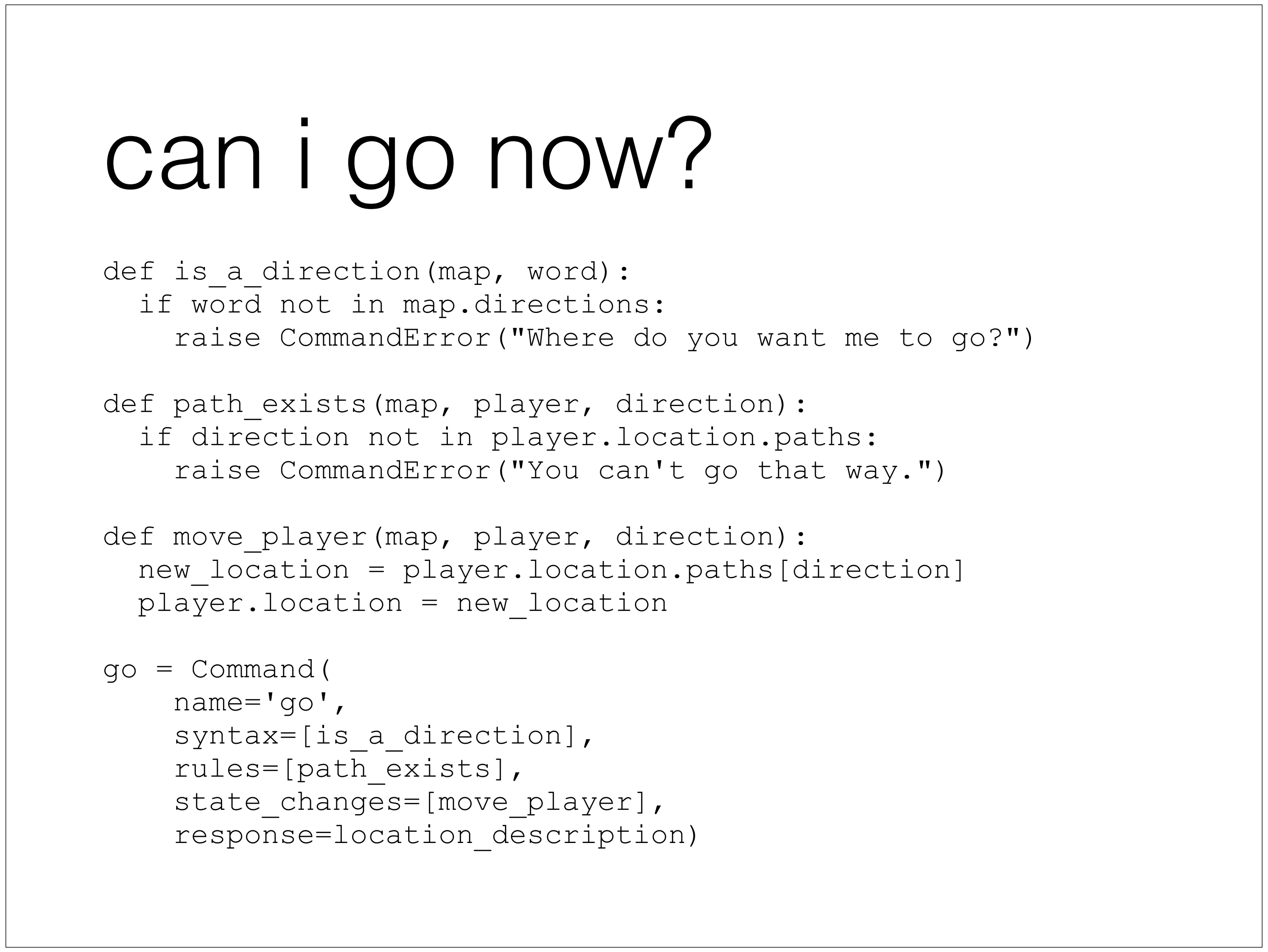
```
go('wand') => wat
go('south') => gotcha

# you are in antarctica
go('south') => "can't"

# you are in vancouver wa
go('south') => "welcome to pdx"
```

If I type go south and there is a room to the south, the go command should move me south and then return the description of my new location. If there is *not* a room to the south, the go command should not move me.

> execute command

- **syntax check** – “do these arguments make sense? I can’t exactly take east.”
- **logic check** – “can I do that? I can’t go west if there’s a brick wall in that direction.”
- **(optional) state change** – “ok looks like I can take that book for you, let me modify your inventory accordingly”
- **formulate response** – “I’ve moved you south, here is the description of the new room you’re in”

22:30 I distilled those requirements down to four responsibilities, and wrote a Command class that is instantiated with helper functions for each of these responsibilities and that, when executed, calls those helper functions in this order.



can i go now?

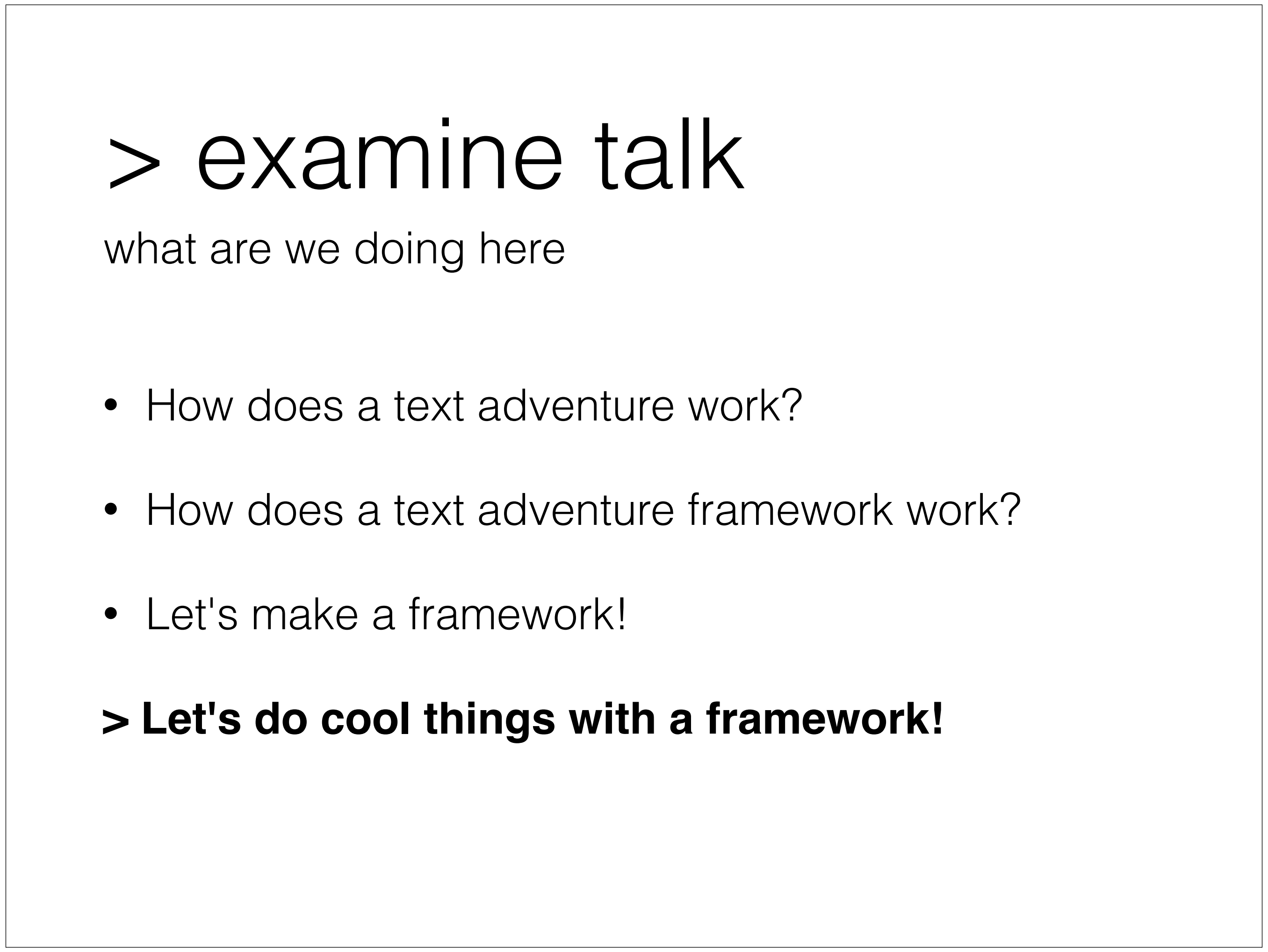
```
def is_a_direction(map, word):
  if word not in map.directions:
    raise CommandError("Where do you want me to go?")

def path_exists(map, player, direction):
  if direction not in player.location.paths:
    raise CommandError("You can't go that way.")

def move_player(map, player, direction):
  new_location = player.location.paths[direction]
  player.location = new_location

go = Command(
    name='go',
    syntax=[is_a_direction],
    rules=[path_exists],
    state_changes=[move_player],
    response=location_description)
```

For example, go looks something like this. Granted, it’s a bit wordy. But now, if I want to include new rules, all I need to do is write a new function and add it to the rules, no sticking logic in stateful classes necessary.

oobleck!

```
def not_in_oobleck(map, player, direction):
  if player.location.oobleck:
    raise CommandError("You are stuck in oobleck!")

go = Command(
    name='go',
    syntax=[is_a_direction],
    rules=[path_exists, not_in_tractor_beam],
    state_changes=[move_player],
    response=location_description)
```

24:40 For example, imagine that you are making a game in which it is possible for players stuck in oobleck, which prevents them from moving. The only change needed to add this bit of game logic is another rule on the `go` command that checks for oobleck.



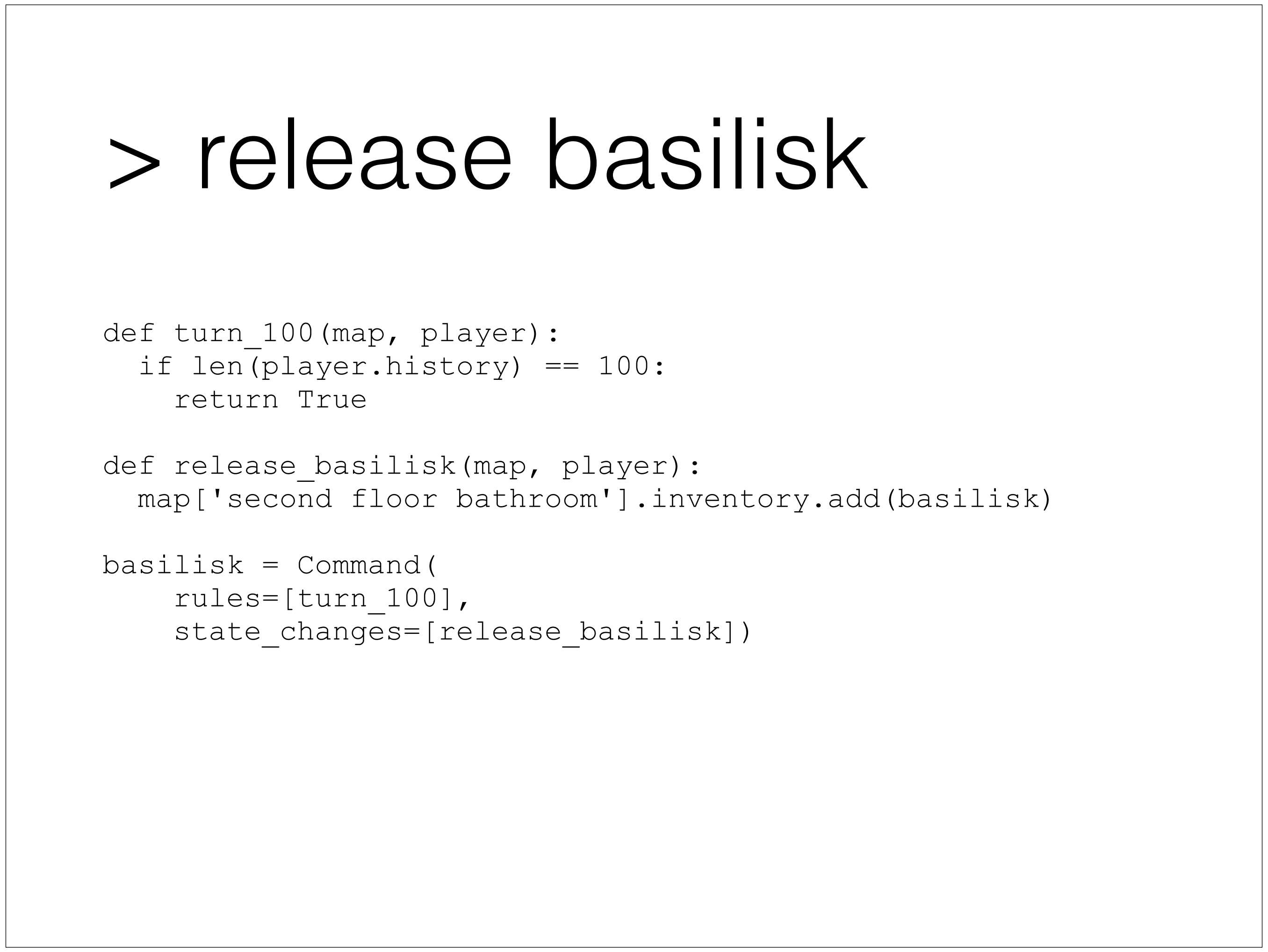
> examine talk

what are we doing here

- How does a text adventure work?
- How does a text adventure framework work?
- Let's make a framework!

> Let's do cool things with a framework!

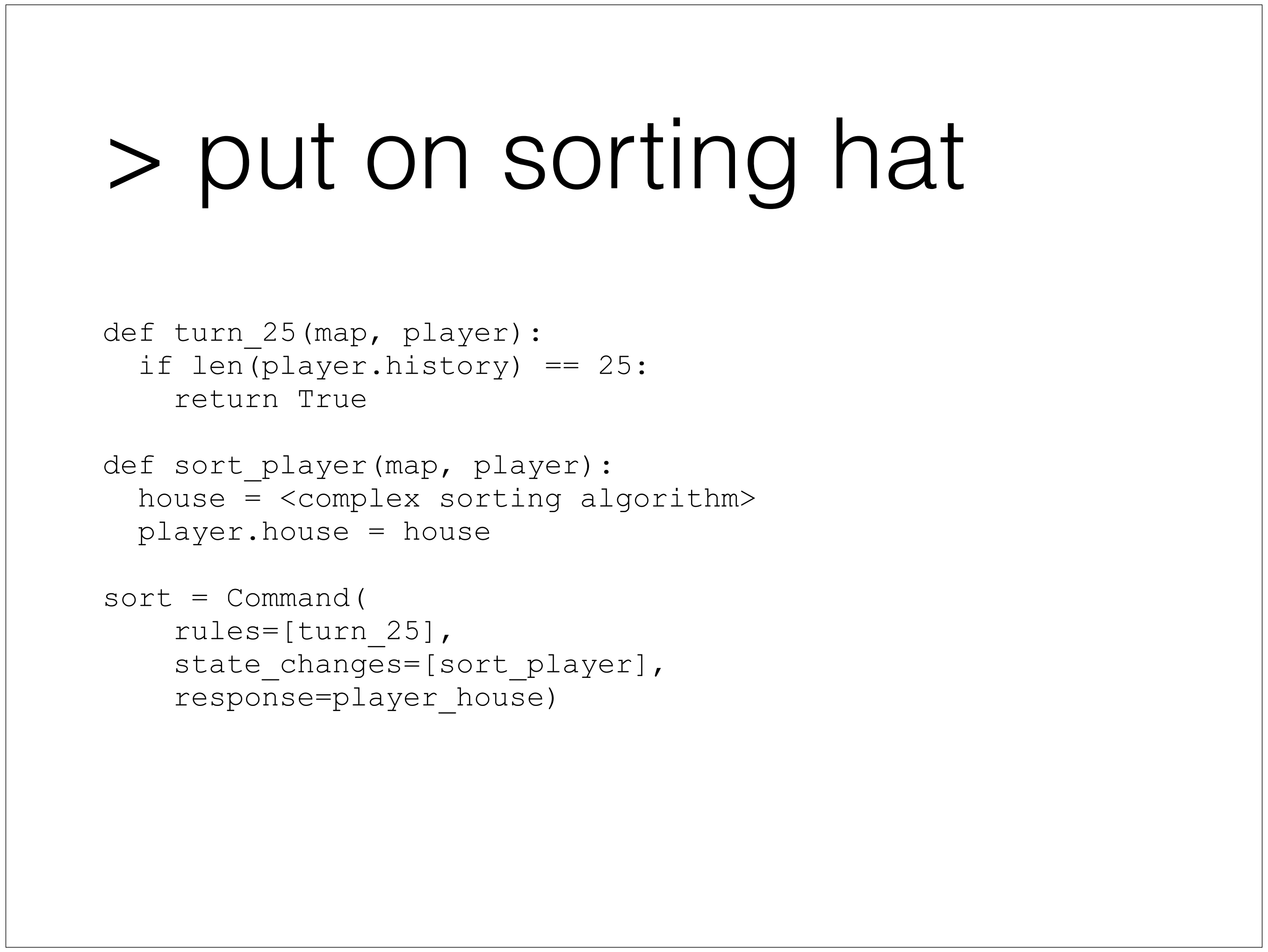
25:00 You made it! Great! Congrats. As a bonus,



> release basilisk

```
def turn_100(map, player):
  if len(player.history) == 100:
    return True

def release_basilisk(map, player):
  map['second floor bathroom'].inventory.add(basilisk)

basilisk = Command(
    rules=[turn_100],
    state_changes=[release_basilisk])
```

Customizable commands can also be used to make things happen at scheduled times in your game! I made the logic handler optionally have a set of commands that it runs after dealing with user input, so you can add commands with rules that allow game state to change once the player has progressed far enough in the game. For example, maybe you want to release the basilisk from the Chamber of Secrets after 100 turns. You'd need a function that, well, releases the basilisk, and a function that return True on the 100th turn. Then just add `basilisk` to the commands that are run after every turn, and you're set for castle-wide petrification!



> put on sorting hat

```
def turn_25(map, player):
  if len(player.history) == 25:
    return True

def sort_player(map, player):
  house = <complex sorting algorithm>
  player.house = house

sort = Command(
    rules=[turn_25],
    state_changes=[sort_player],
    response=player_house)
```

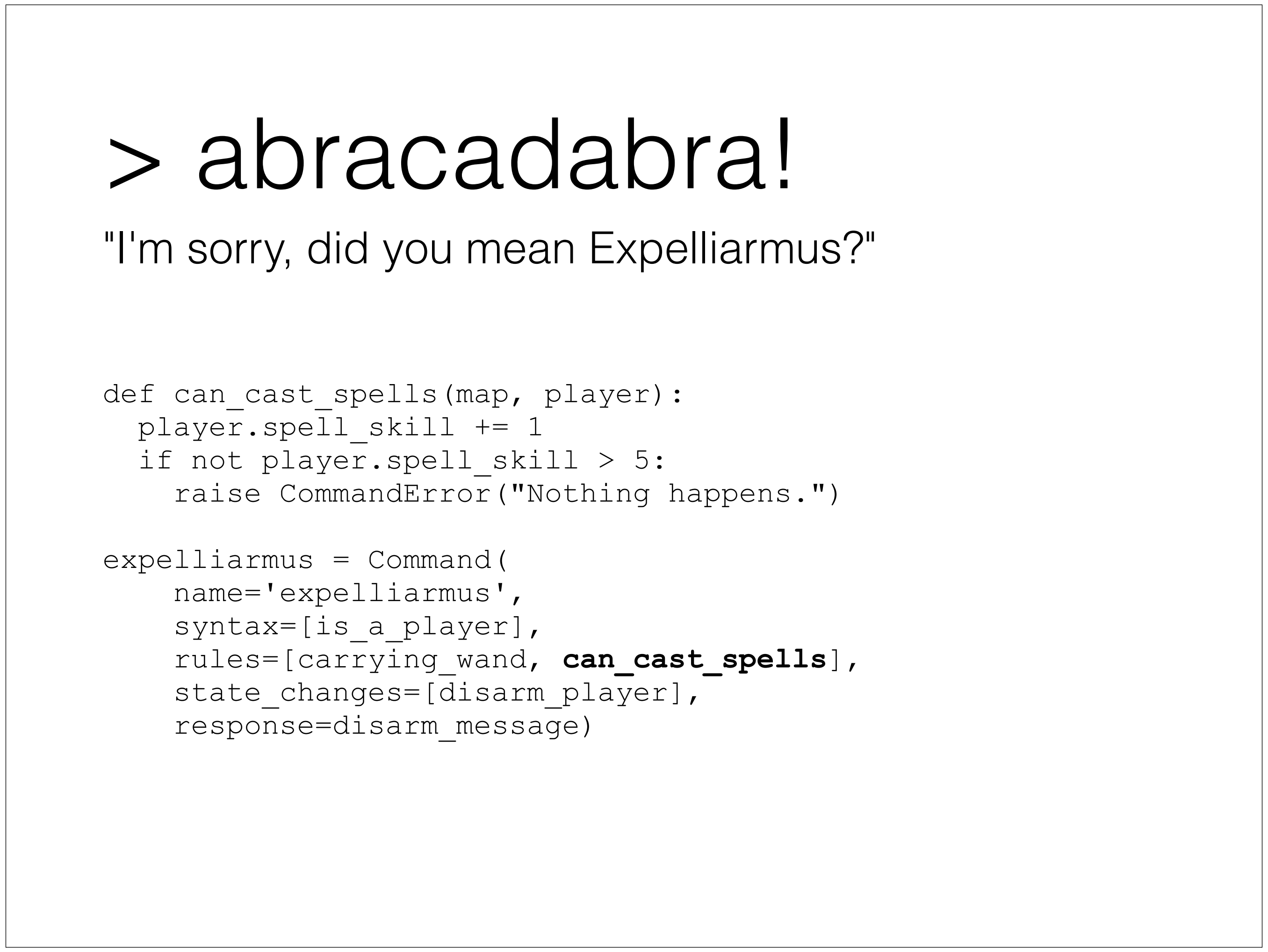
27:30 Similarly, you could decide to sort people after a particular turn. Assuming you have a reasonable algorithm for sorting players based on their move history, you could just add schedule the algorithm to run on a player after a particular number of turns. Sorting is particularly exciting because the house attribute that gets set on the player can impact them at other points in the game (*cough* Sword of Gryffindor).

> abracadabra!

"I'm sorry, did you mean Expelliarmus?"

```
expelliarmus = Command(
    name='expelliarmus',
    syntax=[is_a_player],
    rules=[carrying_wand],
    state_changes=[disarm_player],
    response=disarm_message)
```

Furthermore, it turns out that customizable commands make spells a lot easier to write. For example, you can implement spells just like any other command, but with a rule that ensures that the player is carrying a wand.

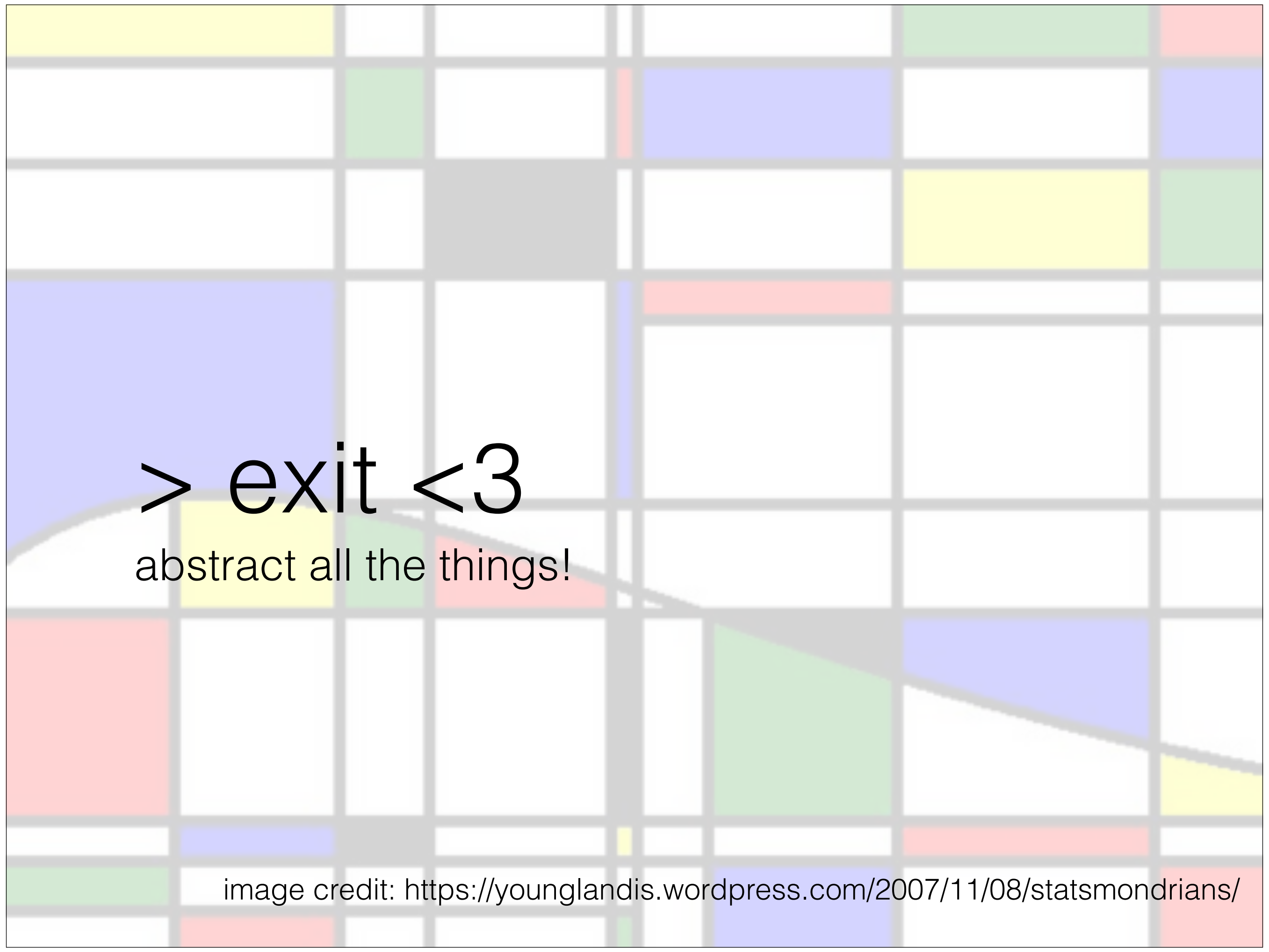


> abracadabra!

"I'm sorry, did you mean Expelliarmus?"

```
def can_cast_spells(map, player):
  player.spell_skill += 1
  if not player.spell_skill > 5:
    raise CommandError("Nothing happens.")

expelliarmus = Command(
    name='expelliarmus',
    syntax=[is_a_player],
    rules=[carrying_wand, can_cast_spells],
    state_changes=[disarm_player],
    response=disarm_message)
```

Finally, you can force players to practice spells before they can actually perform them! Imagine that you increment an attribute on the Player every time they (try to) cast a spell. You can add a rule that will not allow the spell's state change to happen unless the Player has practiced a particular number of times.



> exit <3

abstract all the things!

image credit: https://younglandis.wordpress.com/2007/11/08/statsmondrians/

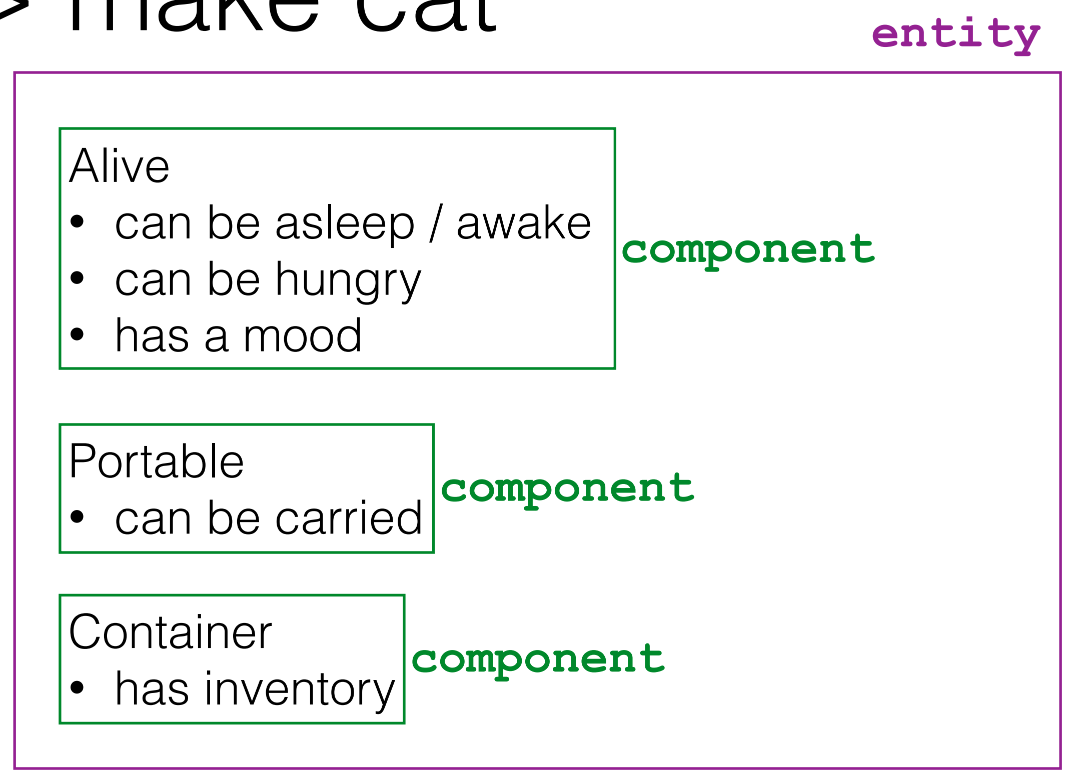
That's about all I got! If I have a point, it is that abstraction makes better UX for everyone--software creators and users alike. On a more personal note, this text adventure adventure helped launch my career--this was the project I paired on for my Recurse Center interview, and I've been learning as a result of working on it consistently ever since. So, if you have a pet project that you're really excited about, build it.

if ye had the chance to change yer state...

customizable stateful things

ECS

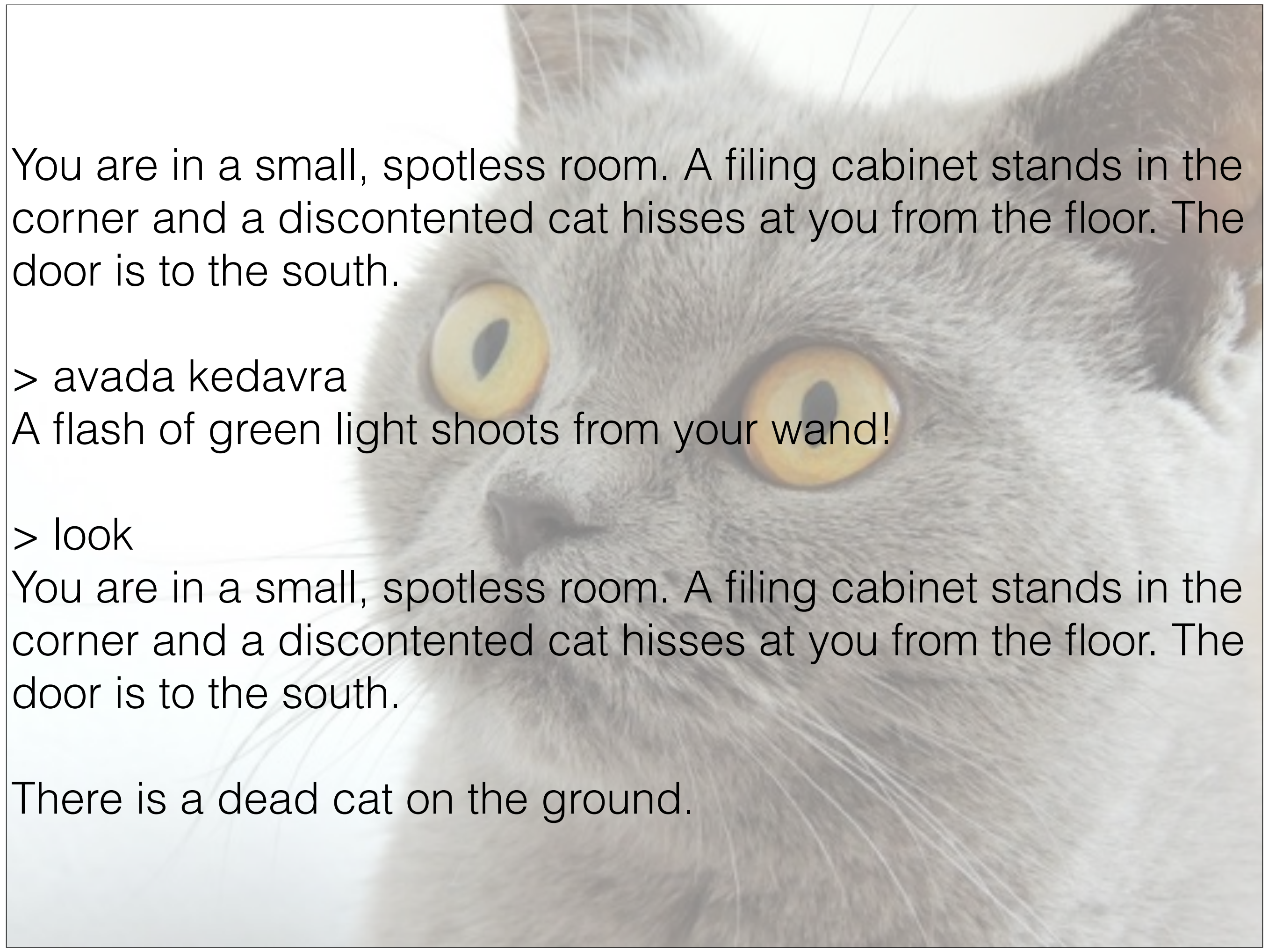
> make cat


entity
Alive
• can be asleep / awake
• can be hungry
• has a mood
component
Portable
• can be carried
component
Container
• has inventory
component

an undead cat

a case study

level of specificity required for description -- anything that changes state should not be in the room description
put it out. people still try to interact with it. limit on writers, moving the UX problem
people try to interact with everything -- UX problem -- they could use some sort of markdown that I parse -- @mention. Markdown creates hyperlinks. what if I as the enginemaker let people write hyperlinks for room descriptions



```
You are in a small, spotless room. A filing cabinet stands in the
corner and a discontented cat hisses at you from the floor. The
door is to the south.

> avada kedavra
A flash of green light shoots from your wand!

> look
You are in a small, spotless room. A filing cabinet stands in the
corner and a discontented cat hisses at you from the floor. The
door is to the south.

There is a dead cat on the ground.
```