Work Summary

The Story

WanderBits is meant to be a game of adventure in the classic sense. Remember Zork? I sure do! I remember learning about it in high school and being completely mesmerized by the entire concept. It was like reading a book with which you could interact directly. I could never get enough of those games!

The game of WanderBits is the story of a young man on a quest to explore his world. We don't know the name of this person, or where he comes from. The world he lives in right now is a bit of a puzzle. It's made of up curiously-connected rooms, but some of the doors are locked and with no key in sight. There must be a key somewhere around? Maybe in that pot over there? Maybe under the carpet? Hmmm??? We must help this poor fellow find his way out of this game so he can continue on with his life!

Game Parts

I decided to base this game around four basic components: a text Parser, various kinds of Actions, various Things to populate the worl, and an Executive to orchestrate the game.

1. **Parser**: A command parser for this type of game needs to support a relatively small number of basic functions. It needs to know about actions for navigating and interacting with the game. It needs to know about things to which the actions are directed. A given command may involve multiple objects, e.g. put key in box. Basic functionality of the `Parser` should be to receive as input text from the user after they presse the Enter key. Idea: parse the text into a list of tokens using something basic like Python's shlex module. Search from the start of the list for a word that matches the name of a command (including aliases). Once found then begin second part of selecting the arguments. White space is no problem. But what about connector words like "the" or "a"? e.g. "Take the apple" should be equivalent to "Take apple" and "Hey, take that apple" or "ehh, grab them apples". I could maintain a list of ignore words and simply skip over them while parsing the user text for commands or arguments. I like that idea a lot, but is it too simple?? I can also make aliases for the plural version of objects by adding an "s" to the end of each object's name.
2. **Actions**: I really like the idea of having a fixed set of arguments for each action command. I can define a base `Action` class that implements basic action capabilities. Then create subclasses for each in-game action. In order to perform its work, the `Action` instance will need access to local in-game items. This could be done by searching through `Thing` instances, starting with the user `Thing`, and then other local `Things` for suitable matches to the argument list. An `Action` makes stuff happen by manipulating in-game `Things` represented by the input arguments. An `Action` must be able to find the proper `Thing` instance(s) that are refered to by a given input argument. These `Things` must be in the local area and also be visible, otherwise the action cannot take place. For example, 'Take apple' may be interpreted as a task to be performed by an instance of `Take` (a subclass of `Action`). `Take` could have a method such as `do_it()`, or `Take` itself may implement `__callable__` and thus be called directly with the input argument. In this case, then input was the word 'apple'. The `Take` action must be able to search the local area for an instance of `Thing` named 'apple'. Once found, `Take` would first ask the apple `Thing` to remove itself from its current container. Then the `Take` action would add the apple `Thing` to the user `Thing`'s container.
3. **Things**: In-game content is to be represented by objects subclassed from a `Thing` base class. Items like an apple or a key could be represented by a `Thing`. But it gets better. A room could be a `Thing`, and even the user could be a `Thing`. A `Thing`-based object would know about concepts such as what other `Thing` is it inside of? A room, a box, or the user's pocket? The act of moving the user from one room to another would involve moving the user `Thing` over to the inside of the next room `Thing`. The basic `Thing` class should handle stuff being inside of other stuff. This means a `Thing` must know about the sizes of `Things` and the size of its own insides. A `Thing` can be destroyed. The set of `Things` inside a `Thing` may or may not be visible to the nearby user. This allows for a box to be opened or closed. Only `Things` that are visible may serve as valid `Action` arguments. A `Thing` is able to describe its appearance when the user looks at it.
4. **Executive**: All the stuff above needs to be attached to something and somewhere there needs to be an event loop running around. Right now I'm thinking there could be an `Executive` module that is the central place where everything comes together. It might be a simple module with a bunch of functions, or maybe it should be its own Class, perhaps yet another kind of `Thing`? This executive could also be responsible for reading text from the user, and then also printing responses back to the user.

Relationships

The `Thing` class is potentially the most complicated one of all. An abstract `Thing` has relationships with other `Things`. The nature of the relationship defines restrictions on whether or not a given `Action` has access to a `Thing`. Relationships may or may not be reciprocal. If `Thing` A is within the local scope of `Thing` B, then `Thing` B is also within the local scope of `Thing` A. However, if A is inside of B, the reverse is *not* possible. Occupying one scope may in addition imply occupying other higher-level scopes.

1. **Intimate**: This scope is used to convey being attached to another item, inside another item, or on top of another item. This implies a physical connection. A `Thing`'s physical location is determined by it's intimate relationship with another `Thing`. An apple `Thing` may be within the intimate scope of a box `Thing`, the box `Thing` may in turn be within the instimate scope of a room `Thing`. An object may only occupy one other object's intimate scope. Multiple objects may occupy a given object"s intimate scope. If A is within B's intimate scope, then B is defined to be within A's *local* scope. Intimate scope is not reciprocal.
2. **Local**: This scope is one layer up from the intimate scope. This scope does not imply physical contact, but it does imply being observable. One may physically manipulate (e.g. take, eat) other `Things` within the local scope. Local scope is reciprocal.
3. **Global**: This scope is automatically implied between all existing in-game `Things`. If a `Thing` is destroyed, it will no longer exist within any other `Thing`'s global scope. Global scope is reciprocal.

These relationship concepts serve primarily as a guidance while developing the game. These terms may not be explicitly realized in any given part of this application.

Thing and Action Subclasses

The `Thing` and `Action` classes are implemented using the Python builtin Abstract Base Class (ABC) module. This was the frst time I used this module in a serious work effort. It's always interesting to learn a new tool! The base class for these two classes serve to define important interface elements for each class. Each subclass for both types are required to implement their own `__init__` method with details unique to each subclass. Inheriting from any base class (not just an ABC base class) makes it much easier to give all downstream children access to the same class properties.

Config File

All data used to describe in-game content is stored within an external YAML config file. I chose this file format since it is very natural for both a person and an application to manipulate the contens. The YAML format also meshes very nicely with Python as these exists direct mappings between data structures in each domain.