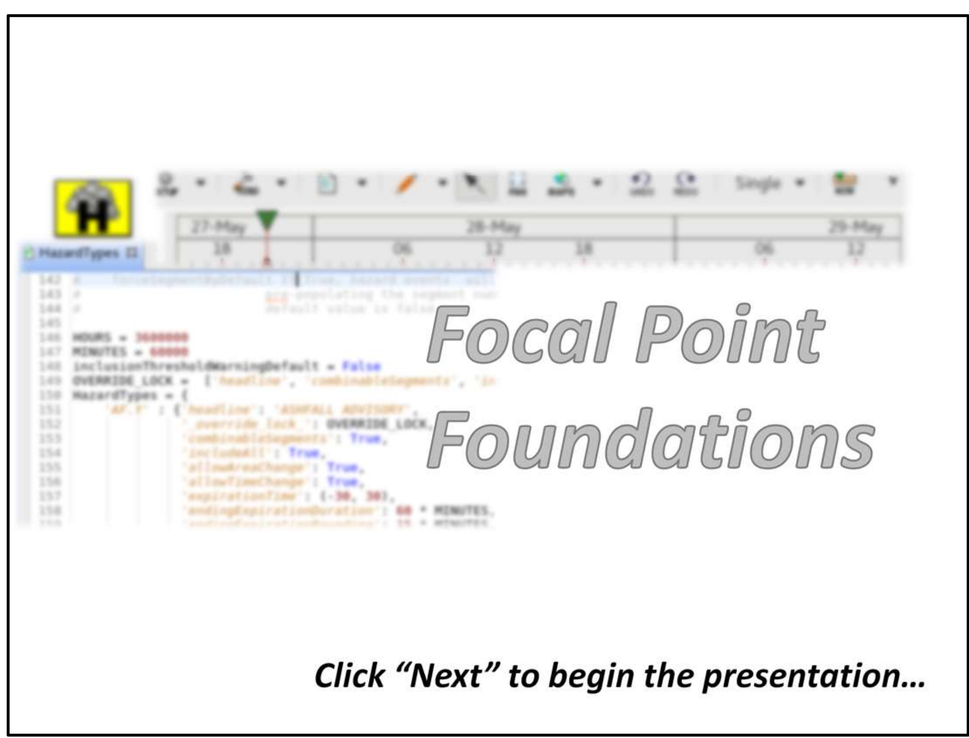


[No audio for this slide]

Warning Decision Training Division

Presenter:
Eric Jacobsen
Course Contacts:
Eric Jacobsen
eric.p.jacobsen@noaa.gov

Overrides

Welcome to the Overrides module, part of the Hazard Services Foundational training course for Focal Points.

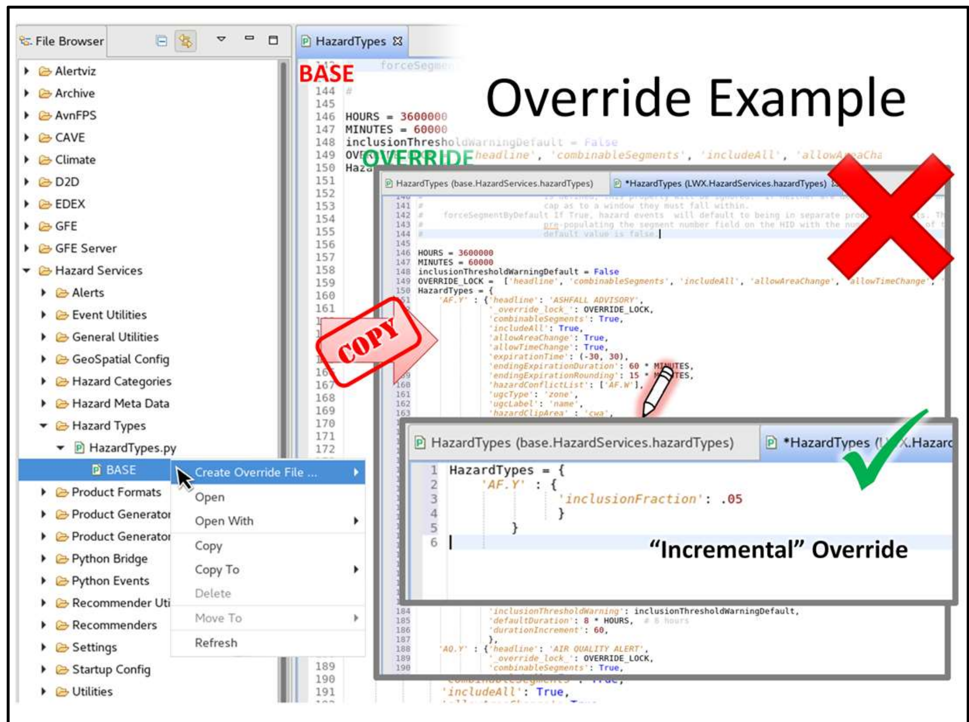
My name is Eric Jacobsen, with the Warning Decision Training Division. If you have questions about this course, or technical problems, please use the contact information listed on this slide.

Overrides: Objectives

After completing this module, you will be able to identify:

- The **motivation behind overrides**
- Incremental Overrides
  - The **definition** of incremental override
  - The **behavior** of lists and dictionaries with incremental overrides
  - The **two main types** of values used with incremental overrides
  - The use of **namespace** to properly format incremental overrides
- Control strings
  - The **purpose** of control strings
  - The purpose of **override_lock**
- Class Overrides
  - The definition of class override
- Benefits of using the **localization perspective** for overrides

These are the objectives for this module. Please take a moment to review them, then, when you're done, click next to proceed with the module.

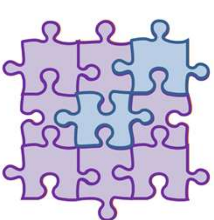


This section covers overrides, the fundamental approach to making changes to baseline Hazard Services configuration files.

To underscore the importance of overrides in Hazard Services, consider the HazardTypes.py file, which is a single large file containing definitions for EVERY possible type of hazard. To make a small change to one of these fields, say to adjust the inclusionFraction which determines how much a hazard must overlap an area to trigger that area's inclusion… copying the entire file to make that one small change would be extreme overkill, not to mention treating every other copied value as a site override as well, despite having no real change.

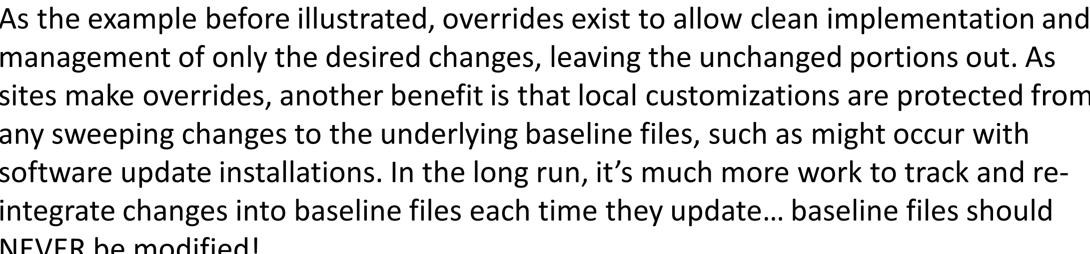
Rather, using focused, incremental-style overrides, we can minimize the file size, more easily maintain our overrides, and prevent overriding whole base files that mostly don't need to be changed. This particular type of override is known as incremental, and it and other similar means of making targeted changes will be outlined in this section.

Override Implementation



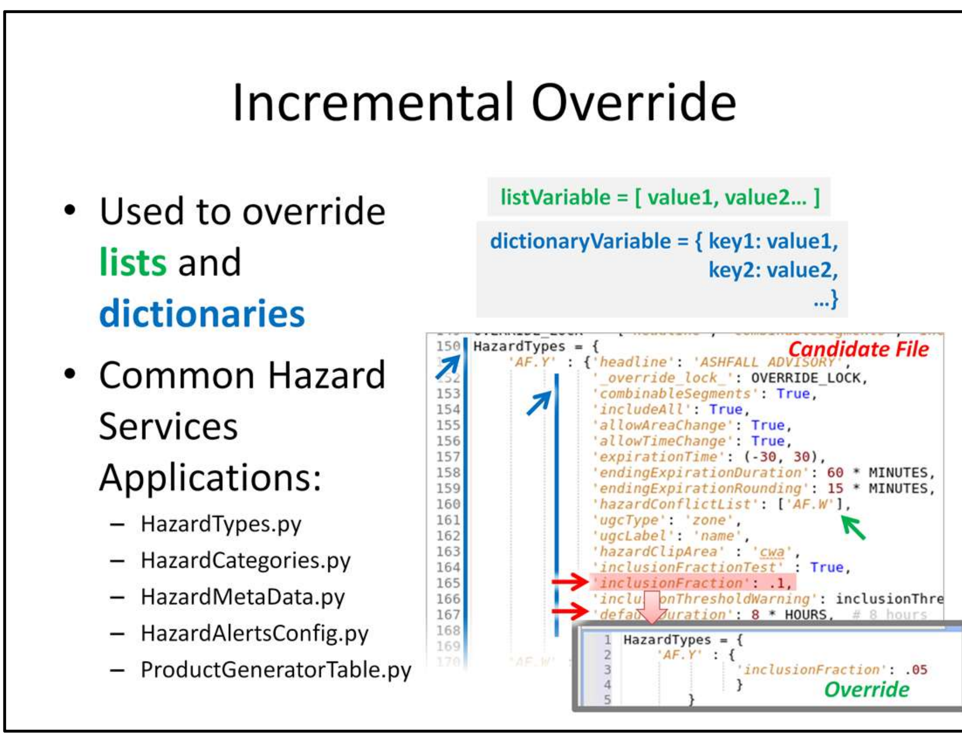
- Purpose:
  - Cleanly manage only desired changes
  - Isolate local modifications from underlying updates
    - NEVER change base files!
- Design:

  - **Conventional:** Maintains awips2 localization level conventions (base, configured, site, etc.)
  - **Targeted:** Extensive & flexible "incremental"-type override capabilities.
    - 2 Significant Types in Hazard Services: Incremental & Class



As the example before illustrated, overrides exist to allow clean implementation and management of only the desired changes, leaving the unchanged portions out. As sites make overrides, another benefit is that local customizations are protected from any sweeping changes to the underlying baseline files, such as might occur with software update installations. In the long run, it's much more work to track and re-integrate changes into baseline files each time they update… baseline files should NEVER be modified!

So how are overrides implemented? First of all, like the rest of AWIPS, Hazard Services still adheres to the standard tiered convention of base, configured, site, and the other levels. Where it departs from the general convention is in its broad adoption of "incremental" and other targeted type overrides. Hazard Services specifically leverages two flavors of this more focused override behavior: one directly referred to as "incremental override"; and the other termed "class override."



Let’s focus on the so-called “incremental” override type first. The “incremental” override applies to configuration files built from Python lists and dictionaries, but not classes or functions (which are covered later). A reminder of what these variable types look like is shown on the screen.

HazardTypes.py, an excerpt of which is shown here, is one important example of a file that is constructed in this manner, using only lists and dictionaries, albeit in a semi-complex nested structure. We’ve highlighted some of the dictionary and list structures in this file with blue and green, respectively. Recall that HazardTypes.py is a pivotal file for setting hazard behavior, including but not limited to: defaults related to its duration; geospatial inclusion criteria; and other parameters which an office might very well wish to customize, through an override. Maintaining changes to this file could very easily become unwieldy, because HazardTypes.py in particular has hundreds of lines of parameters for handling EVERY hazard.

Incremental overrides allow focal points to specifically target small parts of the nested variable structure of these files for change, as we’ll see in the upcoming slides. Just for demonstration, recalling our example override of the Inclusion Fraction for, in this case, the AF.Y hazard, we can see how compact such an override is, relative to the file it overrides. A few other crucial files are listed here which are fundamentally constructed with nested dictionaries and lists like HazardTypes.py, and thus are suitable targets for incremental overrides.

Default Incremental Override Behavior

List:

BASE	SITE	RESULT
myList = [10, 20, 30, 40]	myList =[5] SAME: myList =[5, 10]	myList = [10, 20, 30, 40, 5] ←

Dictionary:

BASE	SITE	RESULT
myDict = { inclusionCheck: False, inclusionFraction: 0.01, extendInTime: True }	myDict = { inclusionFraction: 0.05 }	myDict = { inclusionCheck: False, inclusionFraction: 0.05, extendInTime: True }

- Default merge behavior: Merge/append unique values
- Simple management of only desired change
- Note: Most Hazard Services parameters nested in combination of lists & dictionaries

Generally, how does incremental override work?

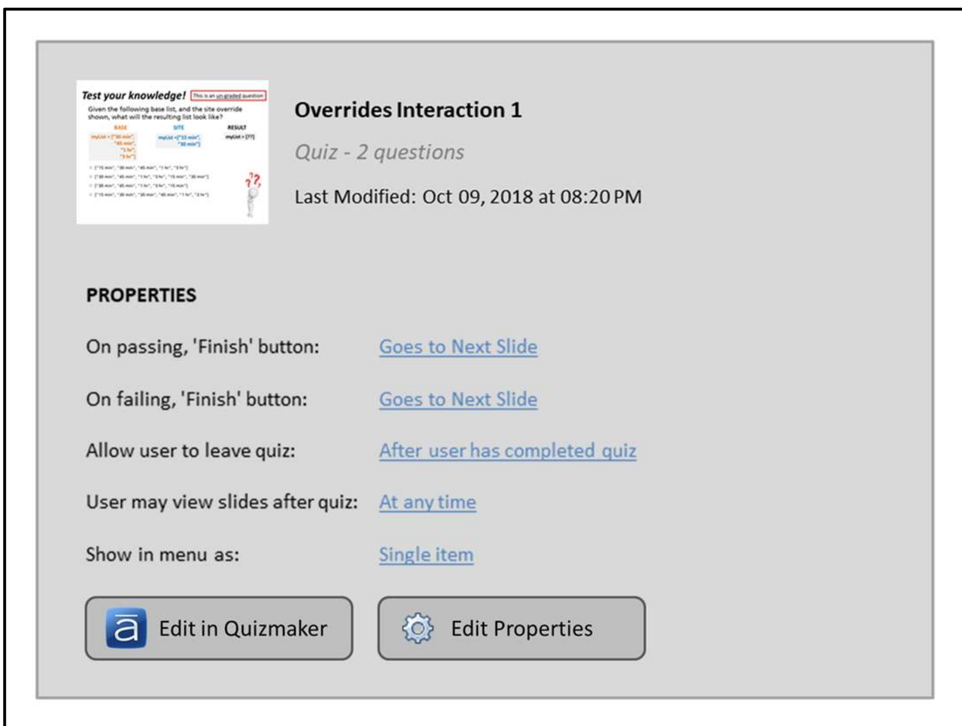
Suppose you have a list in a file at some level (here, base) which looks like this, and we want to override it, say because we wish to include the number 5 as an option. If we create an override file of the same name at a higher level, like site… and in it we reference the same list name (which is how we know what we're overriding), but, unlike the base version, we only include 5 in the square brackets which specify the list values… then, when hazard services loads this variable, it will produce the consolidated result shown at right. The colors are kept to show which level the values came from.

We can see that Hazard Services took the NEW value (which didn't already exist in the original list) and added it to the end of the existing values to make the new list. It's actually important to note that the merge, by default, only APPENDS the new value to end of the list, as opposed to intelligently ordering it. We'll get to controlling that behavior in a few slides. This behavior of simply merging *unique* values is the default behavior of the incremental override. In fact, if we had attempted to override the same variable but included a value that already existed in the base counterpart, such as 10, only the unique, new value, 5, would be added, and we'd get the same result.

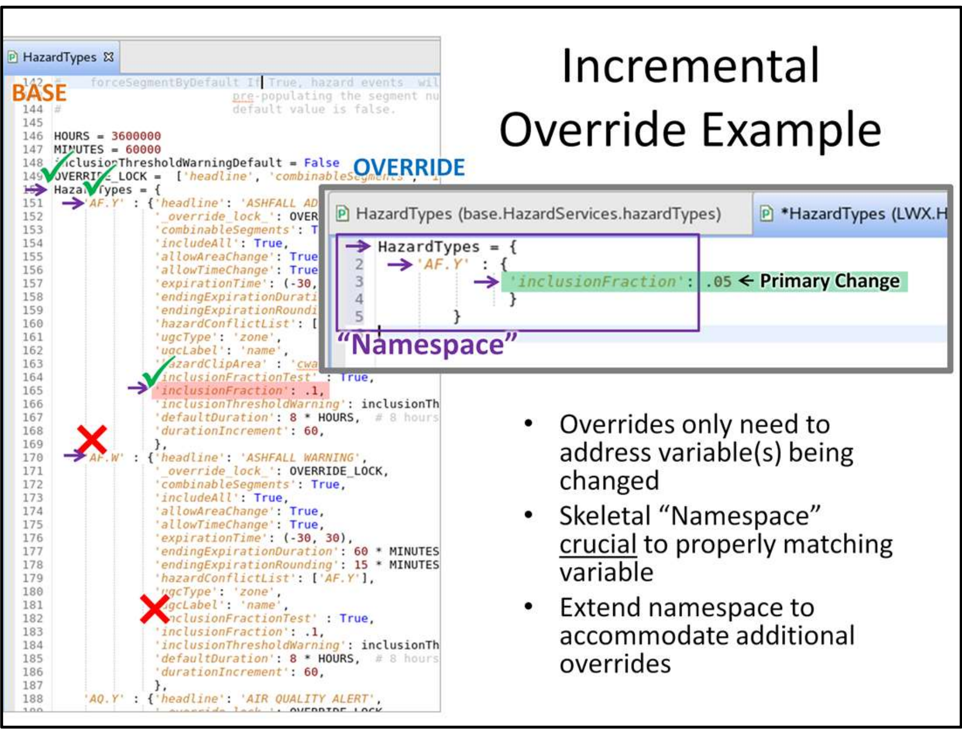
The behavior is similar for a dictionary… when a matching key is found in the same

dictionary name, its value is updated, and reflected in the final, consolidated version of that dictionary when loaded. This simple management of only the parameter needing change is the key advantage of incremental overrides.

Though useful for demonstrating the fundamental behavior, these isolated examples don't reflect that, in most Hazard Services configuration files, a combination of dictionaries and lists is used to specify most parameters. So, how do we navigate potentially deeply embedded variables? We'll consider a real-life example next, after a quick interaction to check your understanding.



[No audio for this slide]



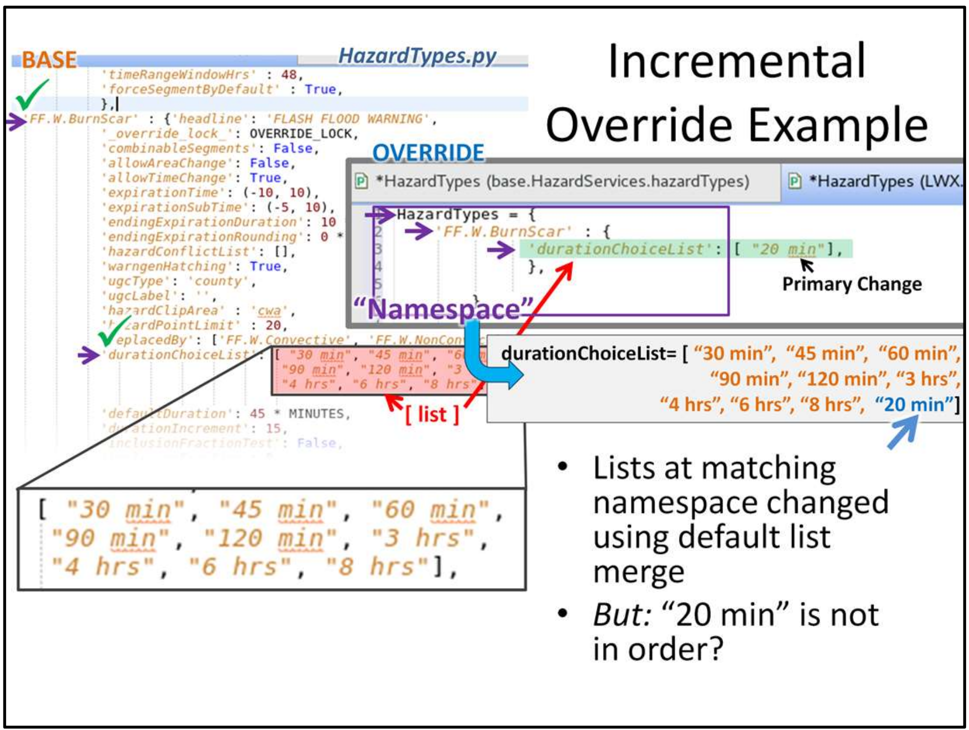
Let's revisit the example this section started with to see incremental overrides in action. The point of this override was to change the inclusion fraction of the AF.Y hazard.

For context, the inclusion fraction is the fractional coverage of a county or zone with a hazard polygon which is needed to trigger that area's inclusion in the hazard… in this case the ashfall advisory. WarnGen focal points should recall similar parameters within the geoSpatialConfig file, and that it is common for offices to have individual preferences for these thresholds. Although we arbitrarily picked the AF.Y hazard at the start of the HazardTypes file, the methodology is identical for any hazard.

The heart of this override is highlighted here, but why is it embedded in multiple layers of other code? Drawing from an important topic in the python overview, the key to performing an incremental override is to know the "namespace" of what you want to edit. Recall that "namespace" is just a fancy way of expressing the "address" of the variable name, including the variables it's embedded in, all the way up the chain to the top of the file. We see here that the inclusionFraction was a key in the AF.Y dictionary, which itself was a key embedded in the HazardTypes dictionary. A valid override for "inclusionFracion" must reflect its position in skeletal form to properly distinguish it for OUR desired hazard (AF.Y), rather than any of the other hazards that also have a parameter with that name.

A quick note about making additional overrides... To be clear, the namespace structure we see in the override is EXTENDED to include any other updates we need to make, NOT replicated. In other words, if we had to make an override to another AF.Y parameter, and even to another hazard, we build these in to the same namespace by extending it as needed, as opposed to replicating the structure for every variable.

If you're new to python, it might take some practice before this structure of variables pops out at you, but jobsheets provided with this training are intended to help build familiarity with this.

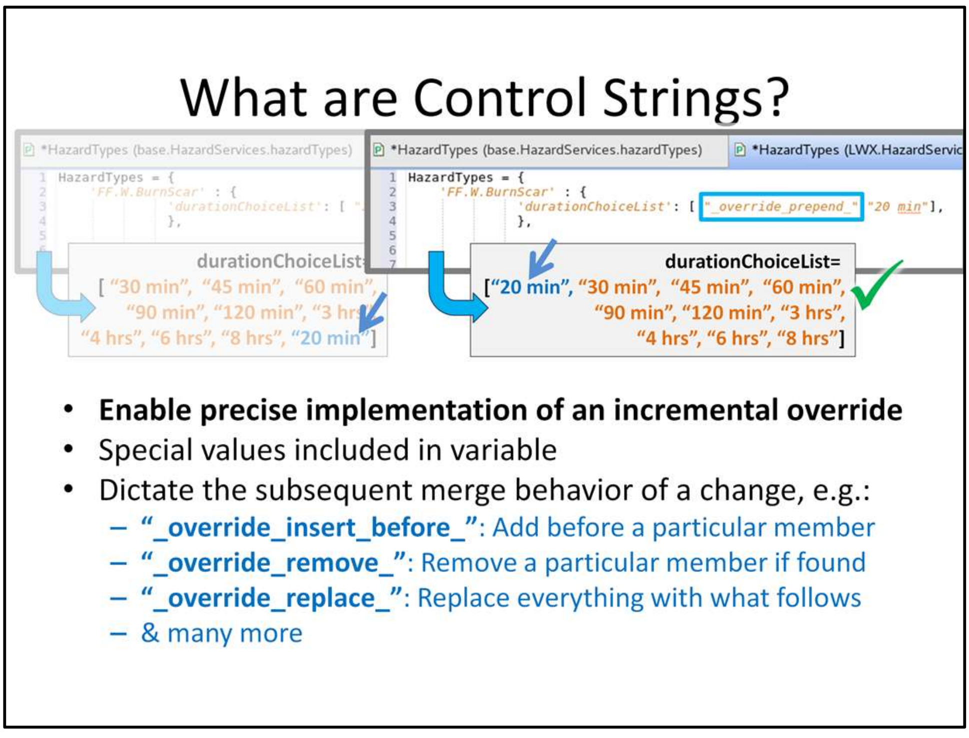


Here is another example of an override in action, this time affecting the durationChoiceList parameter, again in HazardTypes.py. This parameter dictates the choices, in order, of the HID dropdown field for duration, in this case for the FF.W Burnscar hazard type, and which, before any override, looked like this. Changing it will add or remove options for this duration.

As we saw previously, overrides rely on a skeletal replication of the target variable's location to properly define its "namespace." When a base variable at the same "namespace" is found, the incremental override acts on its value.

In our previous example, a simple decimal value was overridden for the inclusion fraction, but this time we find a list at the specified namespace. What does Hazard Services do in this scenario? Just like the default merge behavior we introduced earlier, Hazard Services acts on these list values by merging any new, unique items, by default appending those to the end of the list. Since the "20 min" duration is new and unique, it's successfully appended to the end of our duration list, with the complete result shown.

Though this shows a successful override, the perceptive viewer might recognize a problem… in particular, the failure of this default override behavior to place "20m" at the beginning of the list, its logical place. This conveniently brings us to an important feature of incremental overrides.



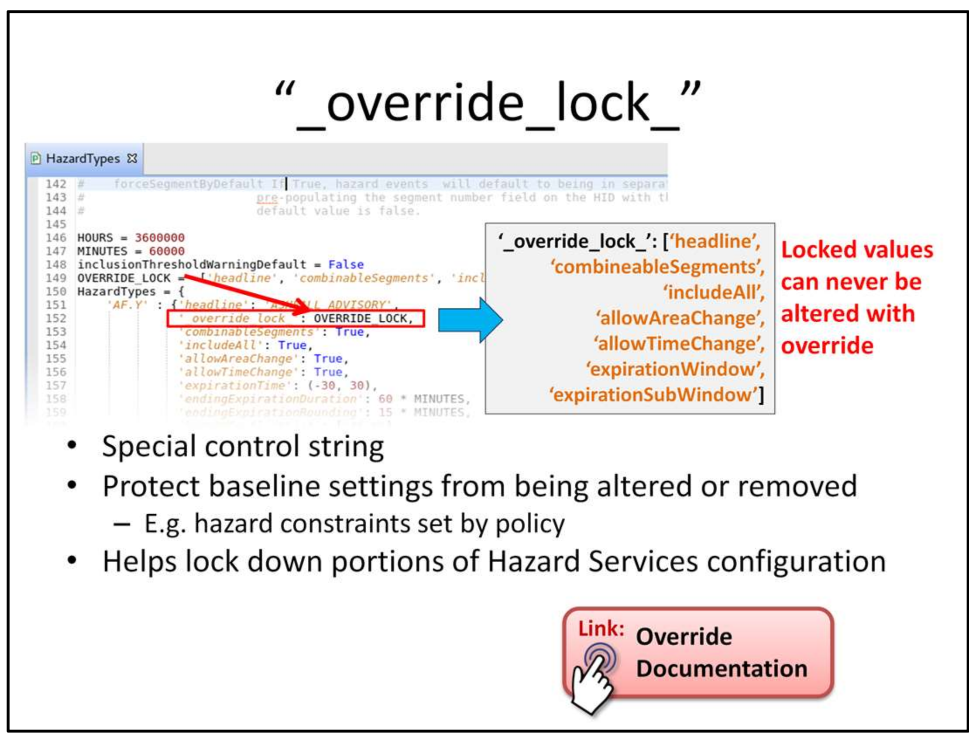
Our previous demonstration of overriding a list showed that the default behavior of Hazard Services's incremental override is sometimes inadequate. To address this, "Control strings" are used. Control strings comprise a powerful toolkit for altering the behavior of overrides, enabling precise implementation of the desired changes.

To demonstrate, let's jump back to the durationChoices override example we used previously. By including, within the override list itself, the control string "_override_prepend_", the merge behavior for our new list item is now to PREPEND it before the base list. So, control strings are, fundamentally, specific string values included within the new list or dictionary variables themselves, which affect the way subsequent values are combined with their base counterparts.

Control string usage can be a tricky topic to grasp at first, and fortunately a comprehensive reference is provided within the focal point guide. However, to convey the breadth of options, a few other examples include the following behaviors: Rather than simply at the beginning or end of a list, to insert a value before a specific member of the base list, use "_override_insert_before_"… Or, Instead of merging, remove the specified value if it's found in the base list using "_override_remove_"… Alternatively, if you use the "_override_replace_" control string, you'll replace the ENTIRE base variable with whatever you've picked for your new values.

The reference on overrides covers many more options. By leveraging one or more

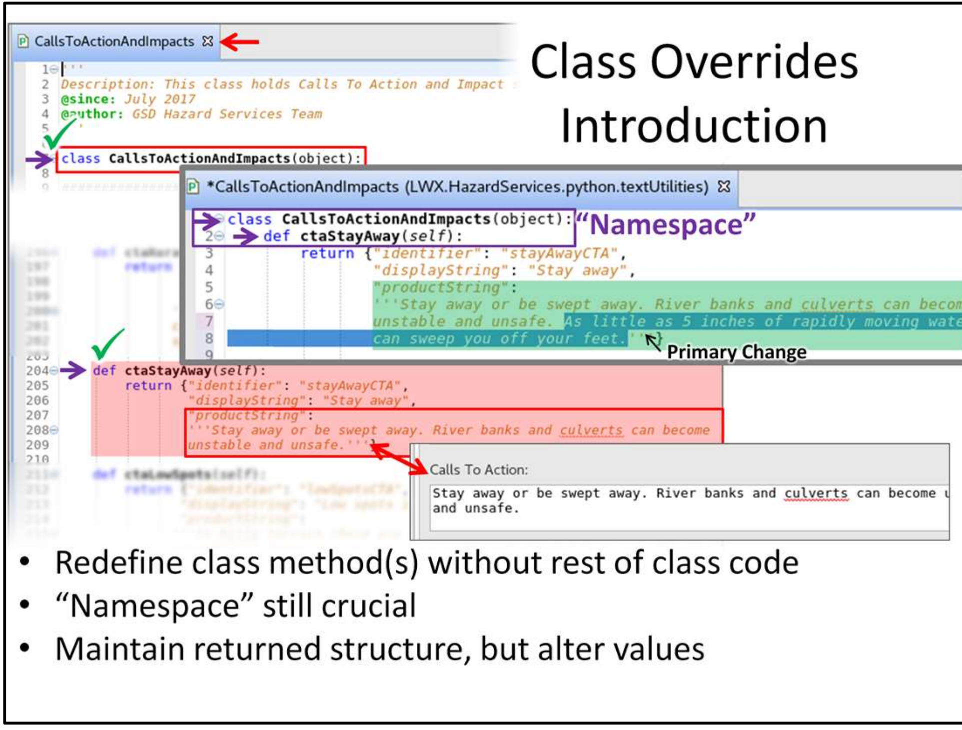
control strings in combination, almost any override behavior can be precisely specified.



In the extensively editable world of Hazard Services, one special control string is worth briefly highlighting… "override_lock," used only in base files, prevents edits to any lists or dictionary keys which it references.

Focal points will not incorporate this into any of their overrides, but they may encounter this lock for certain hazard parameters which are fixed by directive, or which should never be changed, thus preventing their modification.

Again, for more detailed information on this control, and a few minor variations of it, focal points are referred to the override documentation.



The second main type of override which is extensively used in Hazard Services is class-based overrides.

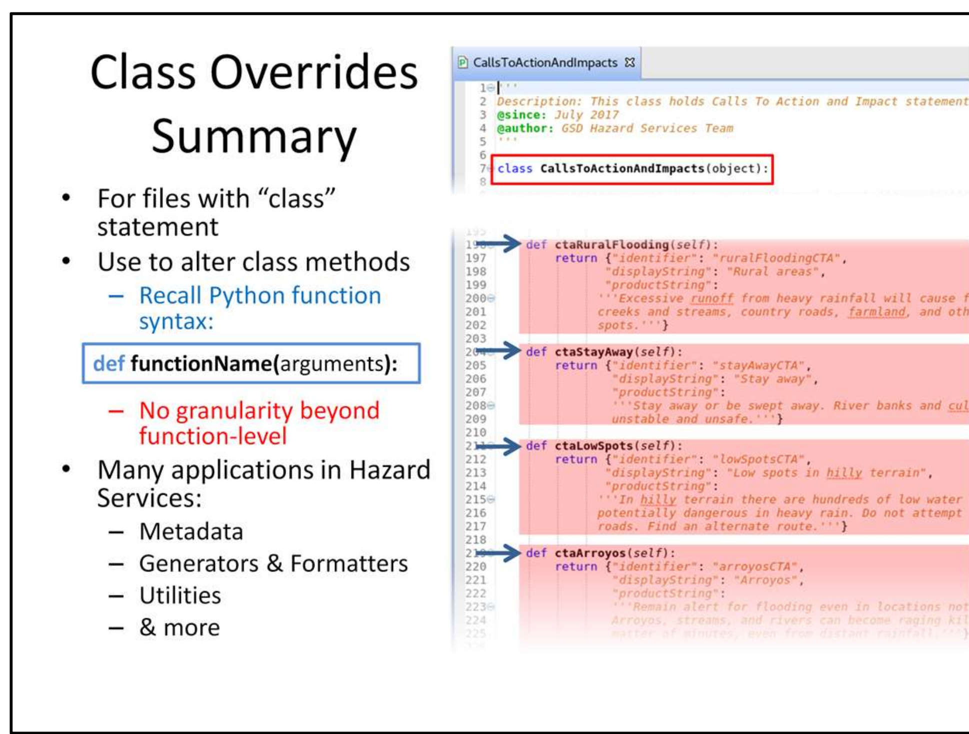
To introduce this type, consider the following example… Instead of basic lists or dictionaries, we're now concerned with changing the behavior of a python function (or method, since it's embedded in a class), called "ctaStayAway". This method, which happens to be embedded in an important utilities file for specifying calls to action and impacts, called ctaImpacts.py, defines the specific phrasing and label for the "Stay Away or be Swept Away" call to action.

We can see it first on the HID for a sample hazard as a call-to-action choice, labeled by its 'displayString' property. Upon previewing the product, this CTA gets translated to full phrasing, this time specified by its "productString", as can be seen in the Product Editor snapshot on screen. It's this product string that we wish to edit, and we can clearly see that this requires editing the ctaStayAway method in the base "CallsToActionAndImpacts" file. There should be some way to modify this method without copying the entire file, right?

In fact, there is, simply by re-defining just the method in question in an override version of the same file. The concept of "namespace" hasn't left us… since this method was included within a class called "Metadata", we need to repeat the same class definition as part of the method override. We can then repeat the expected

return value of the method (in this case a dictionary with 3 key-value pairs) but include the changes we want, such as elaborating on the final CTA phrasing as we've done here.

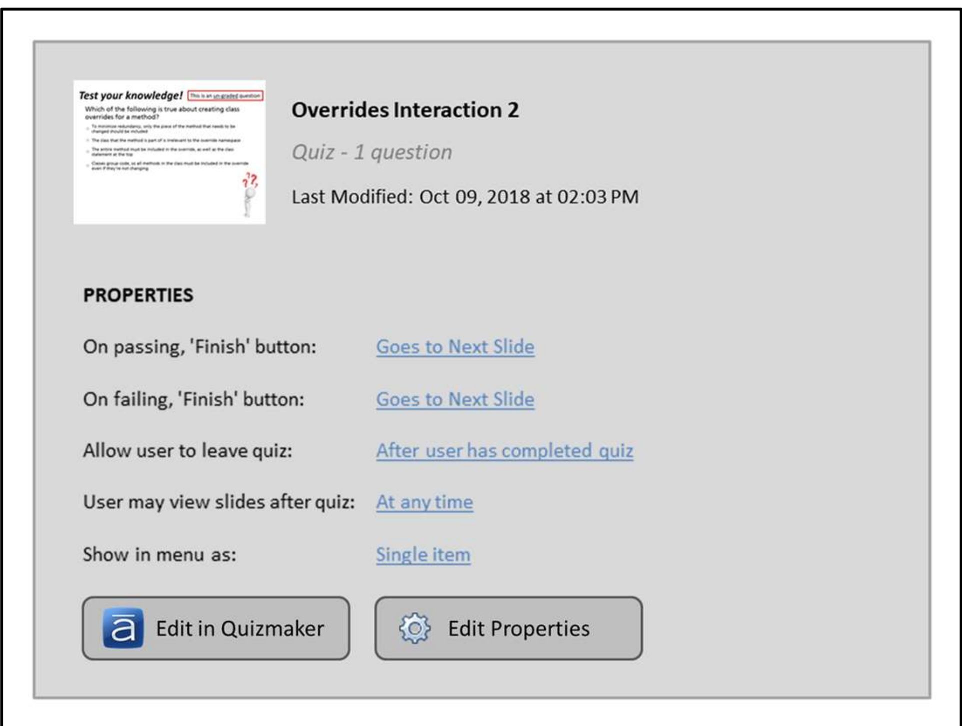
And that's it! Incidentally, if we wanted to include any other overrides of the base file, we could include them as additional methods here, indented within the same class.



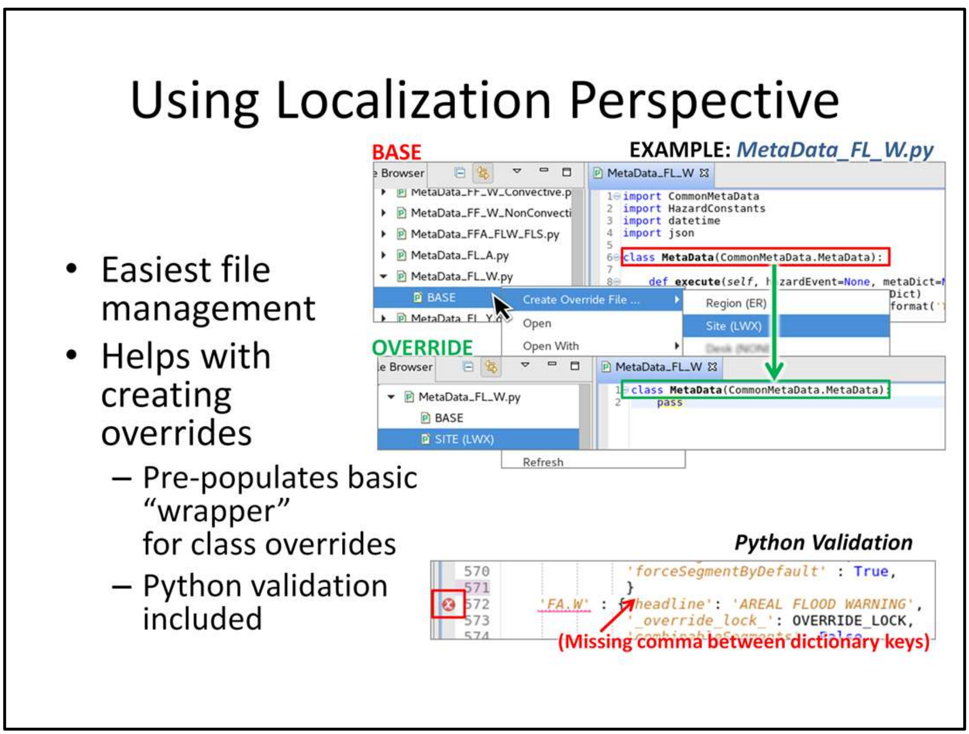
To summarize class overrides, this type applies to ANY configuration file that has a *"class"* definition, like the example shown on the screen. This type of override is primarily useful for altering the methods that belong to a class. Recall from the python overview that methods can be identified by the "def" prefix.

This override lacks the precision of incremental overrides, since we can't just change a part of the method… we have to override the entire method from start to end. But it does allow us to change the behavior of methods, and to the extent that most methods are usually very focused on one task, there is inherent precision in this type of override.

Many core files in Hazard Services use classes, so you'll find yourself using class overrides whenever you need to deal with: Metadata, Product Generators, Formatters, Utilities… and much, much more.



[No audio for this slide]



Though it should go without saying, overrides should be made through the localization perspective for a few key reasons.

First, permissions and file management for the various override levels are most easily controlled this way. In addition, for Hazard Services files, selecting the "create override" option on a file – which is done by right clicking on the file in the Localization Perspective's navigation pane, selecting the override option from the popup menu, and choosing the desired localization level – helps, for class overrides in particular, with pre-populating the class definition that's required as part of the namespace for any methods that follow in the override.

Finally, the localization perspective contains rudimentary syntax checking for Python. As the example on screen shows, where a focal point mistakenly deletes a necessary comma between consecutive dictionary entries, this feature can help with avoiding basic errors that may otherwise be time-consuming to find later, by generally alerting the user to a problem.

Override File Management

- Unprecedented override access
- Typical localization levels:
  - Region, Site, Desk, Workstation, User
  - NO indication/choice given to forecaster
- Best practices for overrides:
  - **Create & thoroughly test at user-level**
  - Properly vet files before "promoting"
  - "Practice Mode" not isolated, but useful for tests
- Benefits of initially restricting localization level:
  - Isolates errors and experimentation from operations
  - Hide development tools and code

Given the unprecedented access to workflow files that comes with Hazard Services, a few best practices for override file management are worth underlining here.

The localization perspective, as should be familiar, makes it easy to create and manage override files, and in particular the localization level at which they reside: region, site, desk, workstation, and user. Like almost all AWIPS configurations, forecasters are given no indication of where in the localization tree a certain behavior is coming from, and no option is available for choosing a different version should an override be broken.

This puts tremendous responsibility on the focal point for vetting the files that are promoted to their site level localization. Despite the site override demonstrations provided in this presentation, ANY override should always be created first at the user level. This isolates its effects, whether negative or simply experimental, from any other users. In addition, this keeps the operational interface free from any development code. Choosing a more restricted localization level for a new recommender, for example, is sometimes the only way of keeping it hidden from and not selectable by a forecaster poking around in their settings. Only after properly vetting files, whether simple or complex, should they be deployed for site or other use, through the "promote" capability in the localization perspective.

Another tool in the focal point trainee's arsenal is Practice Mode. Although

configurations in Practice Mode are NOT isolated from operations because the localization is shared, the ability to use practice events and products can be useful for testing some configurations.

Ultimately, these practices will give focal points the most freedom to master Hazard Services' extensive customizability while minimizing any negative impact through their learning process.

Override Takeaways

- **Do not edit base files**
  - Changes are easier to maintain as isolated overrides
  - Local overrides better protected from build updates
- Two main override types:
  1. **Incremental:** for dictionary & list type file
     - Behaviors to expect:
       - Update value (if target is simple value in dictionary)
       - Merge & append new values (if target is a list)
     - "Control Strings" for greater precision
  2. **Class:** for overriding methods in class files
     - Redefine entire method in override

Overrides, in summary, are a crucial element of proper Hazard Services configuration. Overrides should always be used in place of editing base files, because: it's easier to target and maintain only the desired changes; and especially because they help increase the resilience of site customizations to underlying file changes such as with software updates.

There are two significant override types in Hazard Services: incremental, and class. Incremental handles changes to many key configuration files consisting purely of python dictionaries and lists. By properly using namespaces to target a base variable, it's possible to update its value, or, if that value is a list, Hazard Services by default will create a merged list by appending new, unique values to it. Incremental overrides are also capable of accepting "control strings" for more precision in how this merge occurs.

Class overrides, by contrast, are used to modify the behavior of python methods defined in class files. Such methods shape Hazard Services' behavior to an enormous degree and are likely to warrant customization. Remember, these overrides must redefine the entire method, not just a part of the method.

Override Takeaways, 2

- Skeletal "**Namespace**" crucial to matching variable
  - **Incremental:** Specify all lists or dictionaries in which variable embedded
  - **Class:** Specify class that wraps method
- Use Localization Perspective
  - class wrappers, Python error checking
- Limiting negative impact
  - **User-level FIRST**, then promote when ready
- Override as LITTLE as possible if you can (avoid huge overrides)

```
HazardTypes = {
    'AF.Y' : {
        'inclusionFraction' : .05
    }
```

OVERRIDE

```
class CallsToActionAndImpacts(object):
    def ctaStayAway(self):
        return {"identifier": "stayAwayCTA",
            "displayString"
```

OVERRIDE

572
573

Continuing with our takeaways… Namespace, or, unambiguously specifying where an override target is located, is absolutely central to proper overrides. When creating an override version of a file… IF it is incremental and consists of only dictionaries and lists, then the override variable must be properly nested in a skeletal structure representing any list or dictionary in which it is embedded in the base file. Class override files, similarly, must start with the same basic class statement as found in the base file, after which methods can be written, so as to be unambiguous about which class the methods belong to.

The localization perspective should be used for overrides whenever possible, not only because it is the easiest way of managing the permissions and conventional override level (i.e. site, user, etc.), but because it offers some assistance with namespaces, particularly for class overrides where it will pre-populate the class definition to wrap subsequent methods. And also, because it offers basic Python code validation to help avoid simple but costly syntax mistakes.

We've also emphasized that ANY new files or overrides should first be created at a user level. This critical best practice isolates errors and experimental code from operations, until the code is ready for promotion to broader access.

Ultimately, focal points who can learn to leverage incremental and class overrides to safely change as LITTLE as absolutely necessary of the baseline behavior will be well-

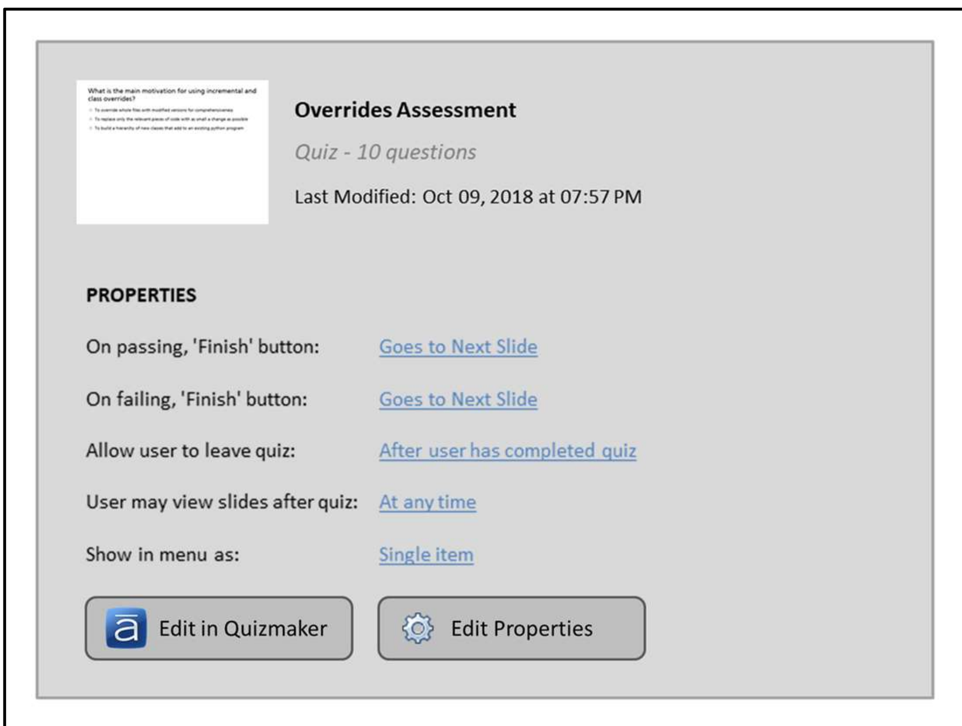
positioned for easier future maintenance of their systems.

Take the Quiz

You're almost finished!

*Click **"Next"** to take the quiz.*

[No audio for this slide]



[No audio for this slide]