

The image shows a presentation slide with a title and a code snippet. The title is "Focal Point Foundations" in a large, bold, grey font. Below the title is a code snippet for "HazardTypes" in a monospaced font. The code includes variables for HOURS, MINUTES, and OVERRIDE_LOCK, and a list of HazardTypes. The slide also features a navigation bar at the top with a timeline and a "Next" button. At the bottom of the slide, there is a text instruction: "Click 'Next' to begin the presentation...".

```
142 | ...
143 | ...
144 | ...
145 | ...
146 | HOURS = 360000
147 | MINUTES = 60000
148 | inclusionThresholdWarningDefault = false
149 | OVERRIDE_LOCK = [ 'HeadLine', 'combinationElements', 'in
150 | HazardTypes = [
151 |   'AP' : {
152 |     'override_lock': OVERRIDE_LOCK,
153 |     'combinationElements': True,
154 |     'inclusion': True,
155 |     'allowRecallChange': True,
156 |     'allowRecallChange': True,
157 |     'expirationTime': [-30, 30],
158 |     'expirationCombination': 60 * MINUTES,
159 |     'expirationCombination': 30 * MINUTES,
160 |   },
161 | ]
```

Focal Point Foundations

Click "Next" to begin the presentation...

[No audio for this slide]

**Warning Decision Training
Division**

Presenter:

Eric Jacobsen

Course Contacts:

Eric Jacobsen

eric.p.jacobsen@noaa.gov



Hazard Services,
Focal Point Foundations Course

Python Concepts

Welcome to the Python Overview 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.

Python: Objectives

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

- The syntax of a Python **list** []
- The syntax of a Python **dictionary** { }
- The syntax of **nested lists** [[]] and **dictionaries** { { } }
- The definition of a **namespace**
- The syntax of a Python **function**
- The importance of **indentation** in Python
- The syntax of Python **classes and objects**
- The syntax of **inheritance**

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.

Python in Hazard Services

- Highlights:
 - Casual syntax supports quick prototyping
 - Extremely versatile
 - Supports object-oriented programming
- Used in Hazard Services for:



Python

- Hazard definitions and settings (*lists/dictionaries*)
 - GUI construction (*metadata & megawidgets*)
 - Product generation & formatters
 - Tools & some data access
 - General configuration
- ... & more

Java/Other

- Primary interface function
- Remainder of data access and storage
- Communications with system

Hazard Services almost exclusively uses Python for its localization and configuration. This means that users of legacy applications, such as RiverPro and WarnGen – which had their own templating languages – will need to learn at least the basics of Python to be successful in configuring their products. Python’s casual syntax, extreme versatility, and support for object oriented programming are highlights of its usefulness.

In Hazard Services, Python code is used in everything from hazard definition files, to graphical interface construction, to actual product creation and more. Java and other languages remain important for interfacing with AWIPS mainly in the background, but are minimally relevant to Hazard Services configuration.

Hazard Services file organization in fact is so infused with Python that this lesson is placed before other topics, in order to introduce some key concepts and syntax before they appear in those later modules.

Topics Covered

- Recognizing:
 - Lists []
 - Dictionaries { }
 - Nested Lists & Dictionaries { [] }
- “Namespace”
- Functions `def functionName():`
- Python Indentation
- Classes & Inheritance `class className()`
- Related Resources



This module alone can't prepare everyone to work proficiently with Python, but it can offer a basic familiarization with some of its key practices.

Many of the most basic configurations can be accomplished in Hazard Services simply by understanding how Python uses lists and dictionaries. These variable types are introduced, as well as how they are very often combined in more complex forms through so-called “nesting.” We'll talk about how properly identifying a variable with its namespace relates to crucial configuration actions like overrides. Many more configuration doors are opened through an understanding of functions in Python, which accomplish critical tasks. Python's characteristic demand for clean indentation will be explained, and we'll also touch on the somewhat more advanced but very relevant concept of classes and what class inheritance means, since it permeates Hazard Services code.

And finally we'll link to some additional resources for reaching the needed level of comfort with Python.

Python Lists

Position indices → 0 1 2 3 4 5 6 7 8

```
durationChoiceList: ["30 min", "45 min", "60 min", "90 min", "120 min", "3 hrs", "4 hrs", "6 hrs", "8 hrs"]
```

in Hazard Services:

- Group of (ordered) values with common purpose
- E.g: list of allowed hazard durations; call to action or impact choices;
- ...

Let's begin with Python lists.

In this and several upcoming slides, we'll use excerpts, like this, of the HazardTypes.py file, which defines hazard types and some of their key attributes.

The list variable types are comma-delimited, ordered collections of numbers, text, object references, or any mix of things. They can be recognized in code by the square brackets that contain them.

We highlight the durationChoiceList parameter in this file as one example of a list. This parameter, incidentally, populates the set of selectable durations in the hazard information dialogue window, an interface we'll cover in much greater depth later.

Returning to our syntax discussion... the value of a list is retrieved in Python by giving the variable name, then, in square brackets, the numerical index of the desired item's position, keeping in mind that Python uses zero-index notation for the first item.

In Hazard Services, as demonstrated here, lists are used to group items with a common purpose. For example: the list of allowed time durations for a hazard type, as shown; or, for assembling the list of call-to-action choices in a hazard's metadata file.

Using the example on screen, a reference to the "durationChocielList" list variable name with the index "2," as shown, returns the third item, whose value is "60min," again because python starts with 0 as the index for the first item, 1 for the second, 2 for the third, and so on.

Python Dictionaries

- **Expression:**
 - Wrapped in { }
 - Comma-delimited
 - Pairs of “key”: value
 - Keys are usually text, but paired value can be almost any type (even other lists, dictionaries)
- **To Access:**
 - dictName[“key”] = Value
 - **OR** dictName.get(“key”) = Value
 - FF.W.BurnScar[“allowAreaChange”] = False

```
HazardTypes.py
'FF.W.BurnScar' : {
    'e': 'FLASH FLOOD WARNING',
    : OVERRIDE_LOCK,
    'combineSegments': False,
    'allowAreaChange': False,
    'allowTimeChange': True,
    'expirationTime': (-10, 10),
    'expirationSubTime': (-5, 10),
    'endingExpirationDuration': 10 * MINUTES,
    'endingExpirationRoundoff': 0 * MINUTES,
    'allowAreaChange': False,
    'ugcLabel': 'key': value,
    'hazardClipArea': cwa,
    'hazardPointLimit': 20,
    'replacedBy': ['FF.W.Convective', 'FF.W.NonConvective'],
    'durationChoiceList': ['30 min', '45 min', '60 min'],
    'defaultDuration': 45 * MINUTES,
    'durationIncrement': 15,
    'inclusionFractionTest': False,
    'inclusionFraction': 0,
    'inclusionAreaTest': True,
    'inclusionAreaInSqkm': 0,
    'inclusionThresholdWarning': inclusionThresholdWarning,
    'startTimeIsCurrentTime': True,
    'allowTimeExpand': True,
    'allowTimeShrink': False,
    'ndLatLon': True,
},
```

in Hazard Services: • Store, and later look up, specific named attributes
• E.g: many hazard attributes, product components, many other configuration variables

Dictionaries are a powerful variable type in Python, which allow you to store collections of key-value pairs. They are contained by curly brackets. The set of comma-delimited key-value pairs allow you to associate a textual lookup ‘key’ with a value you want to store, which itself can be virtually ANY type, even other dictionaries or lists.

By using the dictionary name, and passing it the textual key, which is done in square brackets, you can retrieve the particular value you need. Also, one common alternative for retrieving a key value is to use the “.get()” method on the dictionary variable, this time passing the key name within parentheses.

In Hazard Services, dictionaries are used *extensively*, both in configuration files and under the hood. The central HazardTypes.py file, for example, shown here, uses key-value pairs to specify attributes of each hazard, such as whether or not you can extend it in time. Dictionaries are such useful means of storing information, that even the cumulative result of product generation is itself a massive dictionary, which we’ll show in a later section, where attributes can be called to assemble the product.

In the example on screen, ‘allowAreaChange’ is one key within the dictionary variable for ‘FF.W.BurnScar’... we can access it by calling the dictionary name and passing this key in via square brackets, which will return us the associated value, which in this case is “False”.

Nested Lists and Dictionaries

- Lists and dictionaries can have entries which, themselves, are other lists and dictionaries
- Example:
Lists [] within
Dictionary { }
(HazardTypes.py)

Example Nested Access:

FF.W.BurnScar['durationChoiceList'][2] → "60 min"
Accesses "durationChoiceList" in dictionary, then... *Access item from within list with position index*

```
HazardTypes.py
FF.W.BurnScar: {'headline': 'FLASH FLOOD WARNING',
               'override_lock': OVERRIDE_LOCK,
               'combinableSegments': False,
               'allowAreaChange': False,
               'allowTimeChange': True,
               'expirationTime': (-10, 10),
               'expirationSubTime': (-5, 10),
               'endingExpirationDuration': 10 * MINUTES,
               'endingExpirationRounding': 0 * MINUTES,
               'hazardConflictList': [],
               'warningHatching': True,
               'ugcType': 'county',
               'ugcLabel': '',
               'hazardClipArea': 'cwa',
               'hazardPointLimit': 20,
               'replacement': ['FF.W.Convertive', 'FF.W.NonConv...'],
               'durationChoiceList': ["30 min", "45 min", "60 min"],
               'defaultDuration': 45 * MINUTES,
               'durationIncrement': 15,
               'inclusionFractionTest': False,
               'inclusionFraction': 0,
               'inclusionAreaTest': True,
               'inclusionAreaInSqkm': 0,
               'inclusionThresholdWarning': inclusionThresholdW...
               'startTimeIsCurrentTime': True,
               'allowTimeExpand': True,
               'allowTimeShrink': False,
               'rotationAtLon': True,
               },
```

Before we leave the topic of lists and dictionaries, it's important to know how they can be combined into more complex structures.

As mentioned, since the values within a list or dictionary in Python can be almost anything, then they can certainly also be another list or dictionary. When a list or dictionary is included as a value within a parent list or dictionary, we can refer to them as "nested".

Revisiting the burnscar hazard type definition from before, we can now point out that the list of duration choices we saw earlier is, in fact, nested in a bigger dictionary for the FF.W.BurnScar hazard type. This means that the "durationChoiceList" is actually one of many keys within the dictionary, and the list is its value.

Accessing list members when they are nested is simply a matter of first sequentially calling the key from the dictionary, which gives us the list, then using list indices to get at any of the list values. In this way we simply use the list and dictionary access tools we already know, to navigate down to progressively deeper layers of a nested variable structure.

More on Nested Dictionaries

- Very common for multi-layered data structure

- To access:

- Pass sequence of “keys” to top-level dictionary

- Ex1:

```
DamMetaData["Duckett Dam"]["riverName"] = "Patuxent River"
```

- Ex2:

```
DamMetaData["Duckett Dam"]["scenarios"]  
["rainyDayPMFFailure"]["displayString"]  
= "rainy day (PMF) failure"
```

```
DamMetaData {  
1  
2  
3 "Lake Barcroft Dam": {  
4   "riverName": "portions of Holmes Run, Backlick Run, and Cameron Run"  
5   "damName": "Lake Barcroft Dam"  
6  
7  
75  
76 },  
77 "Duckett Dam": {  
78   "riverName": "Patuxent River",  
79   "damName": "Duckett Dam",  
80   "cityInfo": "Laurel",  
81   "scenarios": {  
82     "sunnyDay": {  
83       "displayString": "sunny day",  
84       "productString": "If a complete failure of the dam occurs, t  
85     },  
86     "rainyDayPMFFailure": {  
87       "displayString": "rainy day (PMF) failure",  
88       "productString": "If a complete failure of the dam occurs, t  
89     },  
90     "downstreamFloodNoFailure": {  
91       "displayString": "downstream flood no failure",  
92       "productString": "Due to releases from Duckett Dam, a signif  
93     },  
94   },  
95   "ruleofthumb": "Interests below Duckett Dam in the city of Laurel an  
96   "dropDownLabel": "Duckett Dam (Howard, PG, and AA Counties)",  
97   "featureID": "DuckettDam"
```

The technique of nesting is even more common with dictionaries. Dictionaries within dictionaries, as seen in this example from DamMetaData, provide a way of organizing data in structures with multiple layers.

The file “DamMetaData.py,” by the way, is what populates the drop-down for dam choices when executing the Dam-break recommender, as shown, a useful tool for generating pre-defined flash flood warning polygons. This interface can contain multiple dam choices, which is reflected in the file as multiple keys, each a different dam, in the first level of the DamMetaData dictionary.

Looking deeper at one of these, Duckett Dam in turn has its own dictionary with keys for river name, dam name, city info, scenarios, and more. To access one of these keys (like riverName), we again just have to sequentially access deeper dictionary keys, which drills down to the values we need, as shown on screen. Meanwhile, the scenarios key, in turn, is yet another dictionary with, in this case, three key values of its own, each containing their own dictionary. For context, these fields help populate the dambreak scenarios that would have appeared as choices on HID after the dambreak recommender was run.

For each scenario, like the one highlighted, its selectable option in the GUI is labeled by the “displayString” key, which we can see embedded deep down in this dictionary. No matter how deep a dictionary or its key:value pairs, we can access them by drilling

down into the keys.

Nesting offers the benefit of organization at multiple levels, with more data available as you drill down into subsequent layers. In Hazard Services, nested dictionaries with many layers are at the core of how hazard information and products are stored and shared. We'll revisit this idea in later sections.

“Namespace” identifies a variable

```
DamMetaData = {
  "Lake Barcroft Dam": {
    "riverName": "portions of Holmes Run, Backlick Run, and Cameron Run",
    "damName": "Lake Barcroft Dam",
    "scenarios": {
      "displayString": "sunny day",
      "displayString": "If a complete failure of the dam occurs, the dam will be destroyed and the city of Laurel will be flooded.",
      "rainDayPMFFailure": {
        "displayString": "rainy day (PMF) failure",
        "displayString": "If a complete failure of the dam occurs, the dam will be destroyed and the city of Laurel will be flooded."
      },
      "downstreamFailure": {
        "displayString": "Downstream flood no failure",
        "displayString": "Due to releases from Duckett Dam, a significant amount of water will be released downstream."
      }
    }
  },
  "ruleOfThumb": "Interests below Duckett Dam in the city of Laurel and the surrounding area.",
  "dropDownLabel": "Duckett Dam (Howard, PG, and AA Counties)",
  "fastAccess": "Duckett Dam"
}
```

- Namespace: the combination of information which uniquely identifies a variable
 - Similar to an “address”
- Prevents ambiguity when referencing variable names
- Essential to Hazard Services overrides

Example:

```
DamMetaData['Duckett Dam']['scenarios']['rainDayPMFFailure']['displayString']
```

* 'displayString' is ambiguous in DamMetaData without reference to its set of parent dictionaries

Having talked about the multi-layer structure of data which is possible in Python, this is the perfect time to define the concept of “Namespace.”


Like the sequence of keys required to identify a specific ‘displayString’ –within a scenario, within a dam, in the previous slide’s example – namespace simply refers to the combination of information which uniquely identifies a variable. It can be thought of like an address.

Let’s think again about the displayString for a dambreak scenario, which, again, specifies the scenario options in the Hazard Information Dialogue for this type of hazard. Not surprisingly, since there are multiple scenarios to choose from here, each with a displayString, we can clearly see that this key name is not unique in the overall dictionary. So, if we want to access or edit one, how does Python know which value to give us?

What made this displayString unique was that we specified we wanted the one in the “rainyDayPMFFailure” scenario, and, what’s more, within the “Duckett Dam” dictionary, to differentiate it from other dams and their scenarios. By being specific about the dictionaries it was nested in, we prevented any ambiguity in referring to “displayString”.

As obvious as it may seem to fully specify a variable’s location when accessing it, a

less apparent need for namespaces arises with Hazard Services overrides. Though overrides are covered in a later section, be sure to remember that successfully overriding any variable depends on properly including it in the parent dictionaries, lists, or other structures to which it belongs.



PythonInteraction1

Quiz - 2 questions

Last Modified: Oct 15, 2018 at 06:47 AM

PROPERTIES



On passing, 'Finish' button: [Goes to Next Slide](#)

On failing, 'Finish' button: [Goes to Next Slide](#)

Allow user to leave quiz: [After user has completed quiz](#)

User may view slides after quiz: [At any time](#)

Show in menu as: [Single item](#)

 Edit in Quizmaker  Edit Properties

[No audio for this slide]

Python Functions

- **Expression:**
 - Defined through:
`def functionName():`
 - Accepts parameters, performs logic, and returns a value
 - Use to isolate fundamental tasks and avoid repetition
- **To Access:**
 - functionName(parameters)
 - Example:

```
productParts_FFA_FLW_FLS_point(self.productDict)
```

in Hazard Services: • *Many* repeated and critical tasks

```
def productParts_FFA_FLW_FLS_point(self, productDict):  
    @productDict -- dictionary created by the generator  
    @return List of productParts including the given segments  
    # Note: Product Parts for each segment / section may be diff  
    # is CAN_EXP is True if the segment has only CAN, EXP in it  
    is_CAN_EXP = True  
    segmentParts = []  
    for segmentDict in productDict.get("segments", []):  
        segmentParts.append(self.segmentParts_FFA_FLW_FLS_point(  
            for vtecRecord in segmentDict.get("vtecRecords", []):  
                if self.tpc.getVtecAction(vtecRecord) in ['CAN', 'E']  
                    is_CAN_EXP = False  
        )  
    )  
    partsList = [  
        'wmoHeader',  
        'CR',  
        'easMessage',  
        'productHeader',  
        'CR',  
        'overviewHeadline_point',  
        'overviewSynopsis',  
    ]  
    if is_CAN_EXP:  
        partsList += [  
            'groupSummary',  
            'callsToAction_productLevel',  
            'additionalInfoStatement',  
            'nextIssuanceStatement',  
            'CR',  
        ]  
    partsList += [  
        ('segments', segmentParts),  
        'initials',  
    ]  
    return partsList
```

Python functions, like the one shown here, are scripts which run on-demand, and which accept parameters, and return values. A quick note that functions are sometimes referred to as “methods” if they are part of another python structure known as a class, discussed shortly.

Function definitions can be recognized by the “def” (short for define) preceding a function name, and parentheses which declare a comma-separated list of inputs that a function will need throughout the commands that follow. Since functions almost always are expected to compute and deliver some result, the “return” statement is a critical piece which specifies what is “given back” to the user of the function. Used intelligently to isolate and define a fundamental, common task, functions dramatically simplify your code, and help you avoid repetition.

In Hazard Services, functions do most of the heavy lifting. In the code excerpt shown here, Hazard Services defines a function called “productParts_FFA_FLW_FLS_point”, whose purpose is to build a “parts list” to begin the FFA, FLW, or FLS point products. In this case, the useful result is partsList, which we see being returned to whatever process requested that the function be run.

Incidentally, functions are used by writing their name, followed by any needed parameters in parentheses, as shown in this snippet from the Legacy FLW_FLS Formatter script which calls our function on the right to start building an FLW or FLS

product.

Python Functions, Continued

```
def ctaTurnAround(self):  
    return {"identifier": "turnAroundCTA",  
           "displayString": "Turn around, don't drown",  
           "productString":  
           "'Turn around, don't drown when encountering flooded roads. Most flood  
           deaths occur in vehicles.'"}  
}
```

- Function Example: “Turn Around” Call-to-Action
- “Discrete” functions with specific tasks:
 - Easy to reuse in many places
 - Easy to override

```
def getCTA_Choices(self):  
    return [  
        self.ctaFFWEmergency(),  
        self.ctaTurnAround(),  
        self.ctaActQuickly(),  
        self.ctaChildSafety(),  
        self.ctaNightTime(),  
        self.ctaUrbanFlooding(),  
        self.ctaRuralFlooding(),  
        self.ctaStayAway(),  
        self.ctaLowSpots(),  
        self.ctaArroyos(),  
        self.ctaBurnAreas(),  
        self.ctaCamperSafety(),  
        self.ctaReportFlooding(),  
        self.ctaFlashFloodWarningMeans  
    ]
```

Functions are so crucial to python and Hazard Services, that we take a look at another example here.

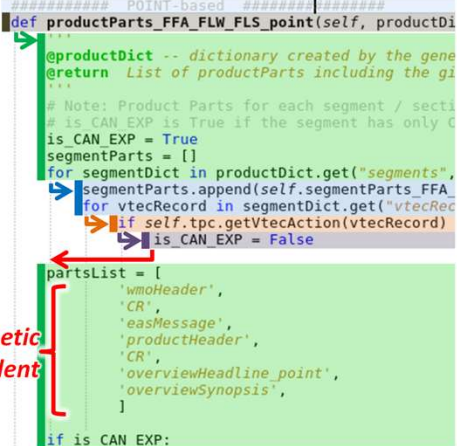
Functions don't need to use complex logic, like the example seen previously. In Hazard Services, each call to action statement in the utilities files is declared with a function, like the one shown here. The only action this function performs is to immediately return a dictionary with different identification and phrase information for the call to action.

Why use a function for this? Defining a function helps isolate a specific, repeatable task, and creates an action that can be called from many locations yet only needs to be maintained in one place. In addition, functions with a specific purpose are easy to change via overrides, a topic covered later in the Hazard Services focal point training.

To again point out how functions get called up for duty, in the Metadata File for FFW Convective, we see this function, and many others, being called, effectively building a LIST of call-to-action dictionaries like the one for turn around.

Python Indentation

- Importance:
 - Crucial to python code structure
 - Indentation changes imply different code block
 - *Required* for function definitions, loops and conditional statements
 - *Optional* extra indentation may help aesthetics
- In Hazard Services:
 - Always be diligent to match expected indentation



```
##### POINT-based #####
def productParts_FFA_FLW_FLS_point(self, productDi
@productDict -- dictionary created by the gene
@return List of productParts including the gi
...
# Note: Product Parts for each segment / secti
# is CAN_EXP is True if the segment has only C
is_CAN_EXP = True
segmentParts = []
for segmentDict in productDict.get("segments",
segmentParts.append(self.segmentParts_FFA_
for vtecRecord in segmentDict.get("vtecRec
if self.tpc.getVtecAction(vtecRecord)
is_CAN_EXP = False

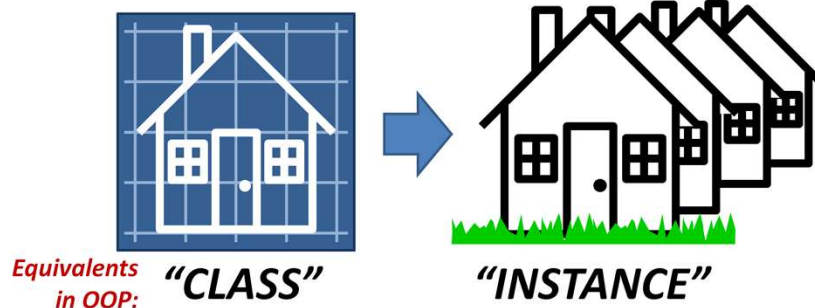
partsList = [
    'wmoHeader',
    'CR',
    'easMessage',
    'productHeader',
    'CR',
    'overviewHeadline_point',
    'overviewSynopsis',
]
if is_CAN_EXP:
```

One of the most distinctive aspects of Python's style is its dependence on clean, consistent indentation for properly interpreted code. Commands with nested code like function definitions, loops, if-statements, and others are strictly grouped according to their level of indentation.

For example, adjacent commands with identical indentation are assumed to belong in the same block, but when Python sees indentation revert or decrease, it assumes the code block has been exited. Two spaces is a common indentation amount for each nested block, but the whitespace amount is less important than consistency throughout the block. In some cases, extra indentation can be used just for aesthetics, such as for ease with reading a set of list values as shown here, though this particular usage is entirely optional.

When editing Hazard Services files, be aware of the indentation amounts to ensure proper grouping or differentiation of your code from what's around it.

What are Python Classes?



- Construct of “object-oriented programming” (OOP)
- Analogous to “blueprints”
- Classes used to create Instances
- Unlimited instances can be created from a class

One of the most important python topics which we have yet to introduce is the use of classes and instances in Hazard Services. Classes are a construct of something called “object oriented programming,” which we mention in passing for anyone interested in diving deeper on their own.

We’re going to use an analogy to take a look at classes before we look at any code. Classes are very much like blueprints, which assemble the instructions and data that’ll be needed, at some point, to create something. Think of how a real blueprint is obviously not itself a house, but shows how to build a house... not only that, but you can reuse the blueprint any number of times to build any number of houses.

In Python, a similar relationship exists and we use the words “class” and “instances” to refer to the blueprint and objects its used to create, respectively. Like the analogy, classes can be referenced *any* number of times to create any number of instances, all generally having the same characteristics.

Python Classes in Hazard Services



MOTIVATION:

Effectively finding and editing methods in a class-based structure

- Syntax:
 - `class ClassName():` creates class and wraps methods below
 - Functions = “Methods” when inside class
 - “self.” points back to class itself
- Classes group methods & data
 - Ready to use over and over
 - Easy to maintain

EXAMPLE: `MetaData_FF_W_Convective.py`

```
5
6 class MetaData(CommonMetaData.MetaData):
7
8     def execute(self, hazardEvent=None, meta
9         self.initialize(hazardEvent, metaDic
10        self._basedOnLookupPE = '{:15s}'.for
11
12        if self.hazardStatus in ["ending", "
13            metaData = [
14                self.getInclude(),
15                ]
16        else:
17            pointDetails = [
18                self.getRiverLab
19                self.getImmediat
20                self.getFloodCat
21                self.getFloodCat
22                self.getFloodRec
23                self.getInclude(
24                ]
25            pointDetails.extend(self.getRise
26            crests = [self.getCrestsOrImpact
27            impacts = [self.getCrestsOrImpac
28            metaData = [
29                {
30                    "fieldType": "TabbedComp
31                    "fieldName": "FLWTabbedC
32                    "leftMargin": 10,
33                    "rightMargin": 10,
34                    "topMargin": 10,
35                    "bottomMargin": 10,
36                    "expandHorizontally": Tr
```

Let’s look a little deeper now, using some actual Hazard Services code. But as we do so, keep in mind the goal is not to get into all the nuances of classes...Rather, the main incentive for understanding classes in Hazard Services, as a focal point, is to better understand where to find and edit methods that get used in a class file, of which there are MANY, and to avoid confusion when those definitions might not be clearly included with the class, but refer to another file.

Here’s a file which specifies “Metadata”, specifically for FFW convective hazards. Metadata, introduced as part of the unified workflow and more thoroughly reviewed in their own module, are crucial components which have direct relation to the unique layouts of the Hazard Information Dialogue for each hazard type, in particular defining what attributes and behaviors are important.

Just to briefly review the syntax we see here... In this file, we see many examples of all the concepts covered so far, including lists... dictionaries... functions... and, of course, indentation. But we also see this crucial “class” entry, with the word “Metadata” following it, which effectively wraps and will “own” all of the function definitions below. By the way, functions which are part of a class file are technically referred to as methods.

Classes are a powerful way of neatly encapsulating many desirable behaviors, through methods that belong to the class, in a way that can be patterned over and over across

unlimited uses, but which are easy to maintain in one place.

As another syntax, note, focal points may often see functions being called with the prefix "self.". This "self" is simply the way a class clarifies that it is calling methods that belong to itself, such as methods defined elsewhere in the file.

Instances of a Class

EXAMPLE: *Metadata_FF_W_Convective.py*

```
5  
6 class Metadata(CommonMetadata.Metadata):  
7  
8 def execute(self, hazardEvent=None, meta  
9 self.initialize(hazardEvent, metaDic  
10 self._basedOnLookupPE = '{:15s}'.for  
11  
12 if self.hazardStatus in ["ending", "  
13 metaData = [  
14 self.getInclude(),  
15 ]  
16 else:  
17 pointDetails = [  
18 self.getRiverLab  
19 self.getImmediat  
20 self.getFloodCat  
21 self.getFloodCat  
22 self.getFloodRec  
23 self.getInclude(  
24 ]  
25 pointDetails.extend(self.getRise  
26 crests = [self.getCrestsOrImpact  
27 impacts = [self.getCrestsOrImpac  
28 metaData = [  
29 {  
30 "fieldType": "TabbedComp  
31 "fieldName": "FLWTabbedC  
32 "leftMargin": 10,  
33 "rightMargin": 10,  
34 "topMargin": 10,  
35 "bottomMargin": 10,  
36 "expandHorizontally": Tr
```

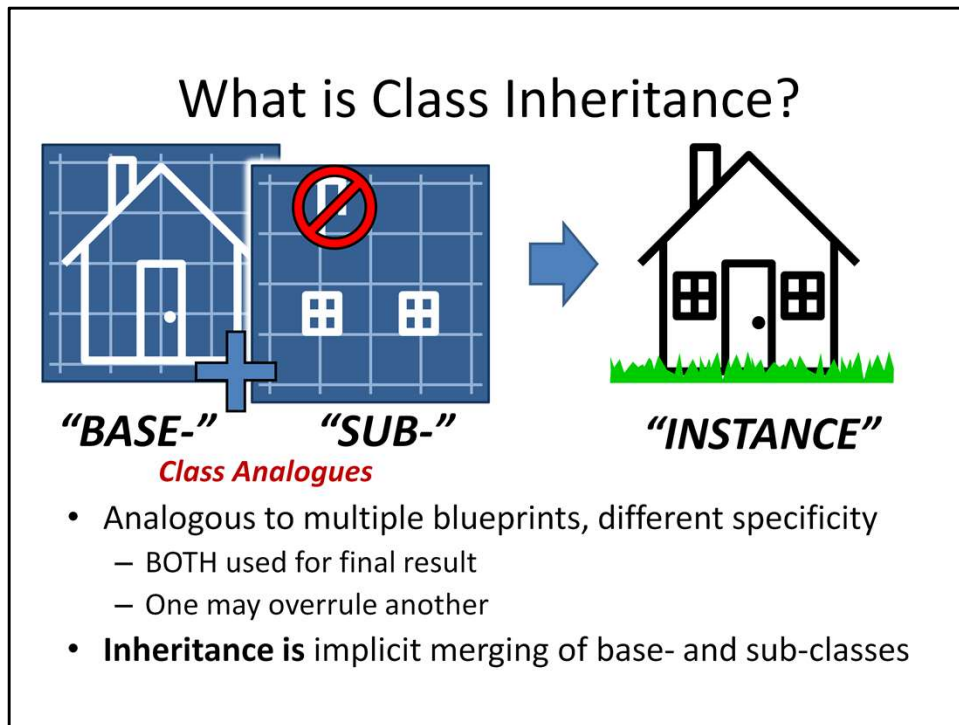
- Metadata class used *each time* HID shown
 - Each “instance” of class has all methods, e.g. “execute()” shown
 - Classes help repeatedly construct complex objects, e.g. HID

Now, returning to the actual role of this file in Hazard Services...

Each time the HID is presented for a hazard of this type, this metadata class is referenced. The code acts as a blueprint for the attributes that may be selected, and their corresponding interface elements on the HID. ANY made-to-order use of this class, referred to as an instance behind the scenes, comes pre-made with all the methods and capabilities in this class file. Each instance, for example, automatically has the “execute” method associated with it, and all the other methods. All of the class methods can be and are used for the logic that’s needed to build the HID for this hazard type, and even more specifically for the circumstances relevant to one hazard.

The general point here is that classes help group code that will be repeatedly used for something, like generating hazard-specific HIDs, and so it’s fairly straightforward to change the blueprint for all instances by finding and editing the relevant method in the class file, like execute here, which is the most central method to HID construction.

But there’s one catch, which is actually another advantage of classes in disguise, BUT which can initially frustrate users who are looking for code in a class that needs editing...



What especially makes classes different from a simple grouping of methods, is something called class inheritance.

Bringing back our blueprint analogy... Inheritance is like using multiple blueprints to build a house where the final house incorporates all the aspects of both blueprints. Why would this be used? Well, one blueprint might specify the general requirements for a house, like that each house has a roof, a door, and a chimney. But a more specific blueprint is used to give the design details of a specific house... like that it also has two windows, and, actually, SHOULDN'T have a chimney. The second blueprint is dependent on the first for the complete house, but also has the final authority in conflicts, such as whether there's a chimney or not.

Python classes can be combined in a very similar way to our blueprint analogy here. A “base” class can specify, in particular, some general behavior, and a derived or “sub-class” INHERITS all of that behavior but may specify its own details, even to the extent of overruling something in the base class.

Class Inheritance in Hazard Services

BASE-CLASS: *CommonMetadata.py* SUB-CLASS: *Metadata_FF_W_Convective.py*

```
class Metadata(object):
    def __init__(self):
        self.logger = logging.getLogger('CommonMetadata')
        for handler in self.logger.handlers:
            self.logger.removeHandler(handler)
        self.logger.addHandler(UFStatusHandler.UFStatusHandler)
        self.logger.setLevel(logging.INFO)
        self.bridge = Bridge()
    def initialize(self, hazardEvent, metaDict):
        self.hazardEvent = hazardEvent
        self.metaDict = metaDict
        # Set current time
        simutime = int((SimulatedTime.getTime().getSystemTime().getSystemTime()) / 1000)
        self.currentTime = int(time.mktime(datetime.datetime.strptime(str(simutime), '%Y-%m-%d %H:%M:%S').strftime('%Y-%m-%d %H:%M:%S')))
        if self.hazardEvent:
            self.hazardStatus = self.hazardEvent.getStatus()
        else:
            self.hazardStatus = "pending"
```

```
class Metadata(CommonMetadata.Metadata):
    def execute(self, hazardEvent=None, metaDict=None):
        self.initialize(hazardEvent, metaDict)
        if self.hazardStatus in ["ending", "ended", "stopped"]:
            self.getEndingOption()
            return
        # Receding water
        self.recedingwater()
        self.rainEnded()
    def getLocationsAffected(self):
        # This method don't have casted be default
        return super(Metadata, self).getLocationsAffected()
    def getAdditionalLocations(self):
        return super(Metadata, self).getAdditionalLocations()
    def applyInterdependencies(triggerIdentifiers, mutablePropertyChanges):
        return CommonMetadata.applyInterdependencies(triggerIdentifiers, mutablePropertyChanges)
    def __init__(self, hazardEvent, metaDict):
        self.hazardEvent = hazardEvent
        self.metaDict = metaDict
        # Set current time
        simutime = int((SimulatedTime.getTime().getSystemTime().getSystemTime()) / 1000)
        self.currentTime = int(time.mktime(datetime.datetime.strptime(str(simutime), '%Y-%m-%d %H:%M:%S').strftime('%Y-%m-%d %H:%M:%S')))
        if self.hazardEvent:
```

As if: All of Common Metadata's methods were copied into FFW hazard-specific Metadata

So, how does inheritance actually relate to focal point duties in Hazard Services?

There are MANY configurable files in Hazard Services which uses classes that inherit from other classes. In short, knowing if your file is a subclass helps you anticipate that SOME of its code may be based on OTHER methods from a BASE class, which means you may have to look for methods in the base class if those require editing.

In fact, in our metadata file example from before, we glossed over the fact that it is a subclass of "CommonMetadata". We know this, because whenever classes are defined with something in the parentheses, as highlighted on screen, this means that they are subclasses, which use the item in parentheses as their base class.

Here's the CommonMetadata file, our base class in this case... Without even looking at any code, you'd be right to expect that common metadata contains more general methods for governing how the Hazard Information Dialogue is created, which are used by many other hazard-specific subclasses.

When it comes to our hazard-specific subclass, simply including CommonMetadata as the base class is almost like COPY-PASTING all of CommonMetadata's code into the FFW specific class file, like we illustrate here... except where any methods conflict, in which case what FFW class says takes precedence. We don't start out writing all our hazard-specific metadata classes like this though, because it's much less efficient to

duplicate all this code in each file, rather than having it in one central and more easily-maintained “commonMetadata” file that can instead simply be pointed to, using class inheritance.

Class Inheritance in Hazard Services



Takeaway:

Some methods might be used by a class even though they belong to another (base) class

```
CommonMetadata.py
class CommonMetadata:
    def getRainAmt(self):
        return {
            "fieldType": "RadioButtons",
            "label": "Rain so far:",
            "fieldName": "rainAmt",
            "choices": [
                self.rain_amount_unknown(),
                self.enterAmount(),
            ]
        }
    def rain_amount_unknown(self):
        return {
            "label": "unknown", "display": "display"
        }
    def enterAmount(self):
        return {
            "label": "enterAmount", "display": "display"
        }
```

```
class Metadata(CommonMetadata.Metadata):
    def execute(self, hazardEvent=None, metaDict=None):
        self.initialize(hazardEvent, metaDict)
        if self.hazardStatus in ["ending", "ended", "e...":
            metaDict = {
                self.getEndingOption(),
            }
        else: # issued/pending
            self.getRainAmt(),
            self.getEventType(),
            self.getFlashFloodOccurring(),
            self.getRainRate(),
            self.getAdditionalInfo(),
            self.getFloodLocation(),
            self.getLocationsAffected(),
        }
    def applyInterdependencies(triggerIdentifiers, mutable
        propertyChanges = CommonMetadata.applyInterdepende
        return propertyChanges
```

... nowhere?

We'll wrap up our brief introduction to classes here with a quick example of a possibly frustrating scenario, which knowledge of class inheritance helps prevent.


A focal point wants to edit the interface piece on a hazard information dialogue (we'll call this a megawidget) for entering the amount of rain received so far. This is populated by the "getRainAmt" method, highlighted here in the FFW metadata file, which by the "self" prefix we initially expect to find as a function definition somewhere in this file... but we DON'T find it. Now what??

It turns out, that this getRainAmt method is actually defined in the commonMetadata class file, and was only being USED by the hazard-specific class. This arrangement makes perfect sense once we know about class inheritance. As we've repeatedly pointed out by now, the hazard-specific metadata is a subclass of commonMetadata, which we knew from the top line where it was defined, and so we should expect, as we see here, that it has access to and can use methods defined in the commonMetadata base class.

Therefore, to edit the method getRainAmt, we have to appreciate that it really lives in commonMetadata, even though our FFW class uses it, probably because it's a communal method with a special place in the base class to allow other hazard types beside our FFW to access it.

Test your knowledge! This is an interactive question

Pick the answer below which best summarizes the Python syntax shown below.



PythonInteraction2

Quiz - 2 questions

Last Modified: Oct 05, 2018 at 11:46 AM

PROPERTIES


On passing, 'Finish' button: [Goes to Next Slide](#)


On failing, 'Finish' button: [Goes to Next Slide](#)

Allow user to leave quiz: [After user has completed quiz](#)

User may view slides after quiz: [At any time](#)

Show in menu as: [Single item](#)

 Edit in Quizmaker

 Edit Properties

[No audio for this slide]

Python Takeaways

- **lists []**
 - Group values with common purpose
- **dictionaries { }**
 - Hold key:value pairs for targeted lookups
- Combinations, i.e. “**nested**” **lists** and **dictionaries**
 - e.g. `{[]}`, `{ { } }`
- Namespace
 - The skeletal “address” of nested variables
- Functions, **def functionName()**:
 - On-demand scripts that perform specific tasks
- Indentation
 - Necessary for grouping code blocks



This section aimed to introduce a range of Python topics necessary for recognizing parts of Hazard Services configuration structure throughout the rest of this focal point training.

We introduced some fundamental variable types and how to recognize them in code, such as: Lists, identifiable by their square brackets, which group values with a common purpose... and Dictionaries, wrapped in curly brackets, which use key:value pairs to group information and allow specific retrieval of desired values.

We also explained that most data structures are a combination of the above types, such as: lists or dictionaries which are themselves values within OTHER lists or dictionaries, which we use the word “nested” to refer to, and which by their complex structure require clear specification of each parameter’s nested address, i.e. “namespace” to effectively navigate.

We also introduced functions, on-demand scripts which accept parameters and return values, and which encapsulate virtually all specialized tasks in Hazard Services python code. And we covered Python’s characteristic dependence on proper indentation to identify blocks of code.

Python Takeaways, Continued

- Classes, `class className()`
 - “blueprints” for complex, repeatable code
 - Group methods and data for common object assembly
- Instances
 - objects created by using class
 - Unlimited number
- Inheritance
 - merging multiple class behaviors together
 - `class subClassName(baseClassName)`
 - Sub-class has full access to base-class methods
- **TIP:** When editing methods in Hazard Services, be aware of class relationships



We also scratched the surface of some more advanced Python topics, for example... Classes and Instances, which are analogous to blueprints and the objects they are used to create, respectively. Classes are extensively used in Hazard Services to neatly group methods and data for repeated use, such as in calling up the metadata requirements for a given hazard type, as we saw. From a practical standpoint, class structures provide one convenient place to review and edit the behavior of certain Hazard Services components, like hazard-specific metadata, and many more.

And we discussed class inheritance, which allows multiple classes to consolidate their methods through a hierarchy of base classes and subclasses, the latter having access to all methods in the former, and also having the authority to overrule the former. For practical purposes, focal points need to know that methods might be used in a subclass which are actually defined by the base class, thus tracking down the code may require moving up the chain of class inheritance to find the relevant file.

Related Python Resources

- Resources for Python:
 - Python web tutorials
 - <https://docs.python.org/3/tutorial/index.html>
 - [Hazard Services Focal Point Guide](#)
- Special Python tools in Hazard Services:
 - MegaWidgets (easy interface assembly)
 - More in “Metadata & Megawidgets” section
 - Resources for help: [Megawidgets Reference](#)
 - Overrides & control strings (precise and efficient alterations)
 - More in “Overrides” section
 - Resources for help: [Overrides Reference](#)

Before we end...this module certainly falls far short of bringing everyone up to speed on basic Python programming, and so the following references are recommended, in addition to the focal point guide, for more practice with these python concepts.

Note that Hazard Services also implements special python tools which were not covered in this section, specifically Megawidgets for interface assembly, and override logic for managing incremental changes to localization files. Since they have special applications, each are covered in a different section of the training, but references for help are included here.


Take the Quiz



You're almost finished!

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

[No audio for this slide]



Python Assessment

Quiz - 14 questions

Last Modified: Oct 15, 2018 at 02:17 PM

PROPERTIES


On passing, 'Finish' button: [Goes to Next Slide](#)


On failing, 'Finish' button: [Goes to Next Slide](#)

Allow user to leave quiz: [After user has completed quiz](#)

User may view slides after quiz: [At any time](#)

Show in menu as: [Single item](#)

 Edit in Quizmaker

 Edit Properties

[No audio for this slide]