LA-UR-17-22137
Approved for public release; distribution is unlimited.

Title: Introduction to Python for CMF Authority Users

Author(s): Pritchett-Sheats, Lori A.

Intended for: Instructional slides intended for wide distribution

Issued: 2017-03-14

(U) Introduction to Python for CMF Authority Users

Primary V&V Seminar
Lori A. Pritchett-Sheats, XTD-PRI

2017 March 7

Abstract

This talk is a very broad over view of Python that highlights key features in the language used in the Common Model Framework (CMF). I assume that the audience has some programming experience in a shell scripting language (C shell, Bash, PERL) or other high level language (C/C++/ Fortran). The talk will cover Python data types, classes (objects) and basic programming constructs. The talk concludes with slides describing how I developed the basic classes for a TITANS homework assignment.

Outline

- Python
  - Introduction
  - Language Basics
    - Data types
    - Functions
    - Memory and References
  - Classes
    - Simple Template
    - Inheritance (Animal Example, `zootopia.py`)
    - Operators
  - Packages
  - Unit Tests
- TITANS Example
- Resources

Python – Brief History

- Advanced (3rd generation) language
- No compiling, an interpreted run-time environment language
  - Memory management handled internally unlike C/C++
- First developed in the late 1980s
  - Version 1.0, 1994
  - Version 2.0, 2000
  - Version 3.0, 2008
  - Last release from the 2.x series is 2.7 and Python 3.6 is the latest (December 2016) release
- Version 2.6 and higher are available on all the LANL platforms and nearly any UNIX-flavored OS ships with Python installed.

Python – Why Use Python?

- Over the past several years, it has established itself as one of the most popular languages in use today.
  - TIOBE, #5
  - IEEE, #3
  - In mix with C, C++, Java and C# as a top 5 language.
  - Crowd source solutions to common needs
- Large (HUGE) open source add-ons especially for scientific programming.
  - Excellent language, combined with NumPy and SciPy, to proto-type numerical algorithms
- Built-in Object Oriented (OO) programming
  - Unlike PERL that attempted to graft OO onto a language that was not originally intended to be OO.
- The learning curve to write code is not as steep as other languages such C, Java, etc.

Python – What I Don't Like About Python

- These are my opinions, but I think many others in the X* community would have similar complaints. ☺
- Python 2.x versions are all backwards compatible, Python 3.x versions are not.
  - They changed the print statement syntax. WHY?!?!
  - https://www.python.org/dev/peps/pep-3105/ - rationale for an explanation on why print was changed.
- Uses indentation (tabs and/or spaces) to create code blocks instead of '{' or statements like 'if/endif, do/end' etc.
  - Can create extremely dense code files, which annoy me. I like white-space.
  - Parser will hit a syntax error and die one line later. *sigh*
- It is a poor substitute for shells like CSH, BASH or even PERL.
  - In my own work, anything that is sequence of shell commands I still use CSH or BASH (>20 command sequence or I want functions).
  - The solution to "How do I capture STDOUT/STDERR output from a system command?" varies quite a bit from version 2.3 to 2.7
- Regular expression handling in Python is clumsy compared to shell languages.
  - You will be annoyed with Python's Regular Expression (RegEx) handling coming from PERL.

Python – Add-on Packages

Matlibplot combined with scipy and numpy can duplicate many basic MatLab features!

- NumPy: Numerical Python
  - N-dimensional arrays
  - Linear algebra routines, Fourier transforms and random number generators.
  - Think GSL with out the licensing hassles.
- SciPy: Scientific Python
  - Requires NumPy
  - Hundreds of common numerical algorithms.
- Other Python packages
  - matlibplot, 2D plotting package (www.matlibplot.org)
  - mpi4py, MPI for Python (mpi4py.scipy.org)
  - Nose, Python unit test framework (github.com/nose-devs/nose)
    - Python has a built-in unit-test module, but nose is a fantastic extension!

Python – Availability and Versions

- The topics covered in this talk are valid for Python 2.7
  - Same version I use for CMF development
  - Be aware that Python 3.x does have differences
- Use binary installers or modules to access different versions of Python
  - Source code for Python exists, but it's a challenging package to boot-strap build.
- Enthought Python Distribution (EPD), software that packages Python 2.x with common third party software like numpy and scipy.
  - Available through a module load on LANL systems:

```
module load python-epd
```

  - Nice GUI editor
- Ananconda, another Python package with third party software, includes Python 2.x and Python 3.x
  - Also available on the HPC platforms through modules:

```
module load python/x.x-anaconda-y.y
```

Python – Everything is An Object

- Everything is an instance of an object
  - Example: A string has multiple methods (actions on the characters in the string). It is more that just a collection of chars!

```
>>> a='this is a string'
>>> print a.count('i')
3
>>> print a.upper()
THIS IS A STRING
>>>print dir(a)
```

Type 'python' at the command line and this will bring up an interactive python session; See the '>>>' prompt.

The command `print dir(anyObject)` will display all the implemented methods for that object.

Python - Numbers

- Four numeric data types `float`, complex, `int` and `long`.
  - `int` is a signed integer the same size as a word, usually 32-bit
  - `long` is 64-bit integer
  - In Python 2.x, math operations between integers results in another integer, i.e. 2 / 3 = 0
  - Most machines, at least the ones we care about, will map float to double precision.
  - Numpy defaults to double precision, but other data-types are possible.
  - In Python3, integers are now longs and integer math returns floats, i.e. 2 / 3 = 0.66666666667
- Good web reference for the different numeric data types. www.tutorialspoint.com/python/python_numbers.htm

Python – Logic and Loop Control

```
if sum > 0.0:
  print 'It is positive'
elif sum < 0.0:
  print 'It is negative'
else:
  print 'It is zero'
```

No switch/ case control, use the else if (elif) syntax

Notice the ':' that ends both the logic and loop declarations. Indentation defines the code block where the logic/loop is defined

```
cnt=0
while cnt < max:
  print 'Hello!'
  cnt+=1

for cnt in range(0,9):
  print 'Hello!'
```

Python – Basic Data Types

- Lists (Array)
  - Ordered sequence (stack)
  - Behaves like arrays you find in other languages
  - Use the brackets [ ] to access particular element of a list.
    - x[0]: First element in list
    - x[5]: Fifth element in list
    - x[-1]: Last element in list
  - Can manipulate any list with following the methods
    - Add element or another list to the end: `append`
    - Remove first occurrence of value: `remove`
    - Remove and return items in the list at some index, default is the last: `pop`
  - Loop Control, use the 'in' operator

```
myList=[3,4,5,6,7]
for x in myList:
  print 3*x
```

Python – Basic Data Types

- Tuples
  - An ordered sequence where the members can NOT be changed (immutable) once defined.
  - Created with () brackets, use the [] brackets to access a particular member.
    - For a one member tuple: `myTuple=('this',)`
  - Loop control is the same as lists
  - Existing tuple can be appended, but not spliced/pop

In a language without the concept of private/public members and nearly everything else IS mutable, tuples come in handy!

```
>>> tupExample=('foo', 'bar', 'hello', 1, 3)

>>> tupExample[0]
'foo'

>>> tupExample[-1]
3
>>> tupExample[1]=3

Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
TypeError: 'tuple' object does not support item assignment

>>> tupExample+=('more', 'stuff')

>>> print tupExample
('foo', 'bar', 'hello', 1, 3, 'more', 'stuff')
```

Python – Basic Data Types

- Sets
  - Unordered collection with unique elements
  - Define using `set(tuple or list)`
    - In Python3 or Python 2.7, can be defined with curly braces
      - `mySet={1,2,100,'this','etc.'}`
  - Need only the unique values in a list, use set
    - `uniq=list(set(myList))`

Set operations are a compact way to compare or unionize two lists without creating duplicate elements.

```
>>>setExample=set(('duck','duck','duck','goose'))

>>> print setExample
set(['goose', 'duck'])
```

Python – Basic Data Types

- Dictionary
  - Collection of pairs (key, value)
  - Define using {} brackets for the collection, using ':' to separate key from value.
    - `myDict={ 'name': 'John', 'Age':30, 'Address': '123 Street'}`
  - Create an empty dictionary with `dict()` or `{}`
  - Key does not have to be a string!
  - Methods to modify dicts
    - Remove (key,value) pairs with `del`: `del myDict['Age']`
    - Add new pairs with []: `myDict['Height']=6.5`
  - Use `iteritems` to loop through a dictionary

Dictionaries are unordered. See OrderedDict in the collections Python module to create dictionaries that will output the keys in the order they were added

```
for key, value in myDict.iteritems():
  print '%s=>%s'%(key,value)
```

Python - Functions

```
def functionname(functionParameters):
  :
  a=[3.0, 1.0, 5.0]
  :
  return something
```

Block of the function definition is indented. Parameters defined in this block are not accessible outside the function (local scope)

Return statement is not required. Python will return a None if not defined.

lambda operator creates an anonymous (no name) function in one line. Used with filter, map and reduce.

```
>>> myList=range(0,10)
>>> print myList
[0, 1, 2, 3, 4, 5, 6, 7, 8, 9]
>>> odd_nums=filter(lambda x: x%2,myList)
>>> print odd_nums
[1, 3, 5, 7, 9]
```

Python – Function Parameters

There are different ways to call `compute`

- Positional

```
compute(1,2,3)
```

- Build a list and pass as positional arguments

```
myArgs=[1,2,3]
compute(*myArgs)
```

- Keyword

```
compute(y=2,z=3,x=1)
```

- Build a dictionary and pass as keywords

```
myArgs={'x':1, 'y':2, 'z':3}
compute(**myArgs)
```

```
def compute(x,y,z):
  return (x*y)/z
```

Define default values of a function as keywords (positional ALWAYS before keyword)

```
def compute(x,y,z=2):
  return (x*y)/z
```

Functions can be nested!

```
def compute(x,y,z=2):
  def average(a,b):
    return (a+b)/2.0

  return average(x,y)/z
```

Python – Functions and *args, **kwrds

```
def compute_area_or_vol(*args,**kwrds):

  if len(args) == 2:
    area=args[0]*args[1]
  elif len(args) == 3:
    area=args[0]*args[1]*args[2]
  else:
    w=kwrds.get('w',None)
    l=kwrds.get('l',None)
    h=kwrds.get('h',None)
    if w is not None and l is not None:
      if h is None:
        area=w*l
      else:
        area=w*l*h
    else:
      msg='Error can not compute area'
      raise ValueError(msg)
  return area
```

```
compute_area_or_vol(40.0,50.0)
compute_area_or_vol(40.0,50.0,90.0)

compute_area_or_vol(w=40.0,l=50.0)
compute_area_or_vol(w=40.0,l=50.0,h=90.0)
```

Listed above are ALL valid ways to call `compute_area`. The `*args` (tuple) and `**kwrds` (a dict pronounced keywords) define variable length argument lists.

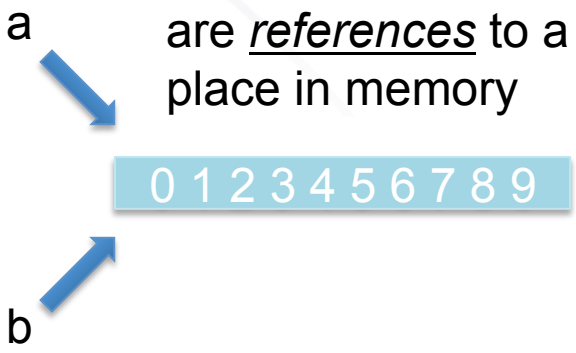
If you have used `printf` in C/C++, you have used variable length argument lists.

Python – References (Memory!)

Try this in an interactive Python session

```
>>> a=range(0,10)
>>> print a
[0, 1, 2, 3, 4, 5, 6, 7, 8, 9]
>>> b = a
>>> b[0]=-100
>>> print a
[-100, 1, 2, 3, 4, 5, 6, 7, 8, 9]
```

What happened to a?!



The variables a and b are *references* to a fixed place in memory

The assignment (=) does not create a new object in memory because lists are mutable, allowed to change in place. Behavior is different for immutable objects (tuples!).
See https://en.wikibooks.org/wiki/Python_Programming/Data_Types for more examples.

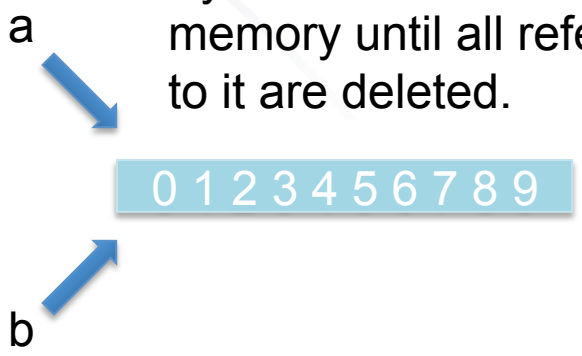
Python – References (Memory!)

Languages like Python that manage memory internally, typically pass references.

When you want a new object in memory with the same values as the original reference, call the “copy constructor”.

In the follow-up presentation, I will point out the ‘copy’ functions that are defined for many of the CMF classes.

I also had tests fail in my TITANS example that were failing because I forgot about this!



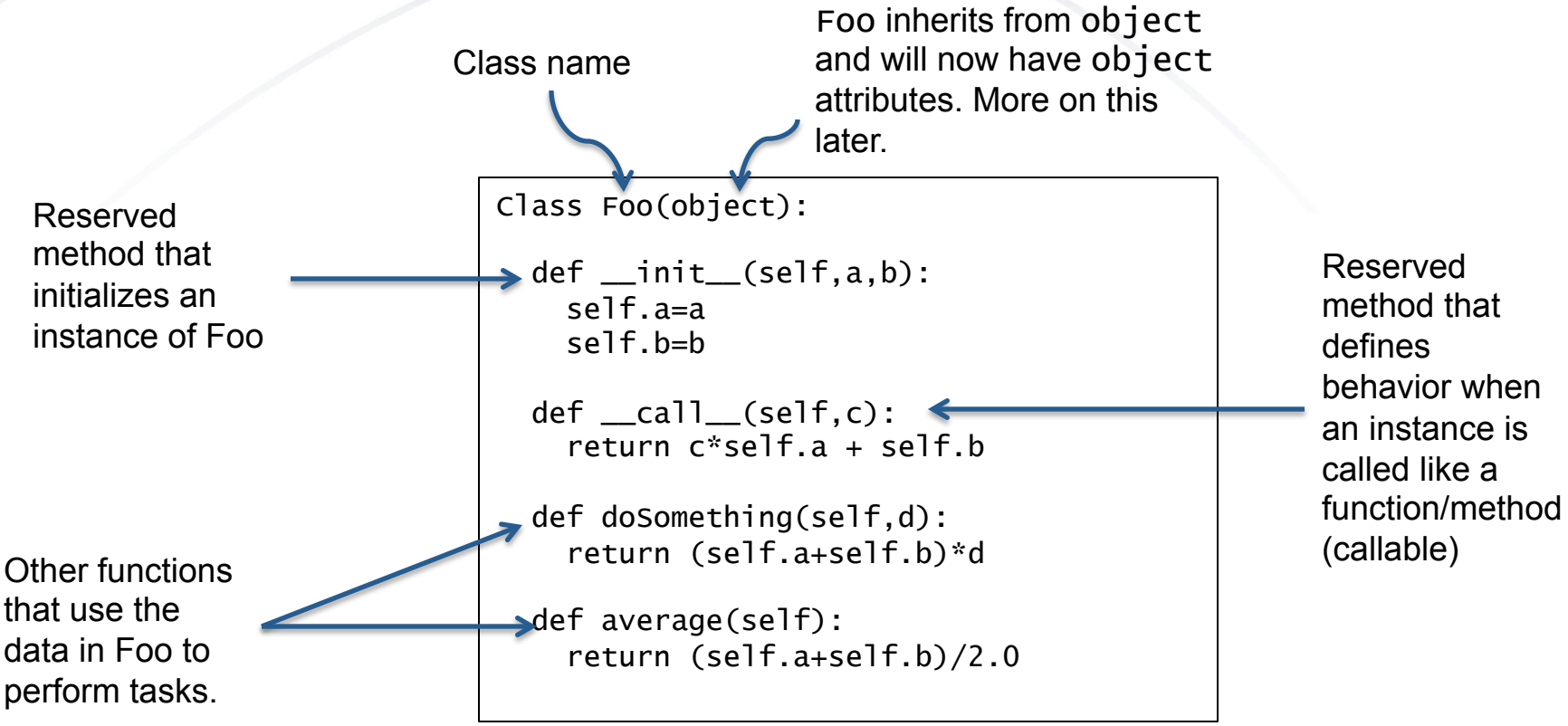
Python will not delete this memory until all references to it are deleted.

Each mutable object in Python has a reference pointer counter that tracks how many references are associated with that bit of memory. If you write code interfaces to other languages like C/C++, you need to be aware of this or you will have memory leaks!

Python Classes

- Why use classes?
  - Organize data and desired software functionality
  - Leverage the Object-Oriented capability of Python
  - Reduce duplicate code
- Based what I have seen of the setup tools for all the teams prior to Common Model Framework (CMF), did not use classes and programmed in a functional pattern.
  - Not surprising, since most developers came from a C or Fortran background.
- When classes have been used, polymorphism was rarely used. Typically a class's purpose was to provide common functions.

Python Classes – A Simple Template



Python Classes – Simple Template

```
class Foo(object):

  def __init__(self,a,b):
    self.a=a
    self.b=b

  def __call__(self,c):
    return c*self.a + self.b

  def doSomething(self,d):
    return (self.a+self.b)*d

  def average(self):
    return (self.a+self.b)/2.0
```

The syntax '`class Foo(object):`' means the `Foo` class inherits from the `object` class. `Foo` will have all the attributes/methods/members defined in `object`.

The `object` class is a built-in class of Python. This allows us to create classes that inherit from `Foo` and use the `super` function, more on that in later slides.

Python Classes – Simple Template

```
class Foo(object):

  def __init__(self,a,b):
    self.a=a
    self.b=b

  def __call__(self,c):
    return c*self.a + self.b

  def doSomething(self,d):
    return (self.a+self.b)*d

  def average(self):
    return (self.a+self.b)/2.0
```

Functions that are members of a class are called methods.

`__init__` is the method called when a 'new' instance of `Foo` is created.

```
f=Foo(2.0,3.0)
print f.a; print f.b
2.0
3.0
```

For any method in the class, the first argument is a reference to the instance calling the method. Typically this argument is labeled '`self`'. This is similar to the '**`this`**' pointer in a C++ class.

Python Classes – Simple Template

```
class Foo(object):

  def __init__(self,a,b):
    self.a=a
    self.b=b

  def __call__(self,c):
    return c*self.a + self.b

  def doSomething(self,d):
    return (self.a+self.b)*d

  def average(self):
    return (self.a+self.b)/2.0
```

Reserved Python methods are named: `__something__`

`__call__` is the method called when an instance of **Foo** is called like a function

```
f=Foo(2.0,3.0)
f(10.0)
23.0
```

Python Classes – Simple Template

```
class Foo(object):

  def __init__(self,a,b):
    self.a=a
    self.b=b

  def __call__(self,c):
    return c*self.a + self.b

  def doSomething(self,d):
    return (self.a+self.b)*d

  def average(self):
    return (self.a+self.b)/2.0
```

Any method defined in the class is called with *instance.methodName()*

```
f=Foo(2.0,3.0)
f.doSomething(5.0)
f.average()
```

Methods can have any number of arguments

Python Classes – A Simple Template

- `f` is referred to as an instance of `Foo`
- Attributes are members of `Foo`. Can be parameters (`a`, `b`) or methods (`average`, `doSomething`)
  - Use `dir(f)` or `dir(Foo)` to return a list of all attributes, including special (`__something__`) and intended to be private attributes (`_doNotUse`).
    - Beware relying on internal attributes!
- Use the '.' for any attribute (parameters or methods)

```
f.doSomething()
f.a
```

- Instances of classes in Python are mutable
  - The creation of a new member is allowed (`f.c=8`) even if `c` was not defined in `Foo` source code.

Python Classes - Operators

- The math operators (+,-,/.*) and comparison operators (==, !=, >, <) are all defined for any Python object through reserved methods.

```
class Foo(object):

  def __init__(self,a,b):
    self.a=a
    self.b=b

  def __eq__(self,other):
    if isinstance(other,Foo):
       return (self.a == other.a) and \
              (self.b == other.b)
    else:
       return NotImplementedError

  def __ne__(self,other):
    return not self.__eq__(other)
```

Operators	Method
==, !=	__eq__, __ne__
<,>	__lt__, __gt__
+, -	__add__, __sub__
*, /	__mult__, __div__

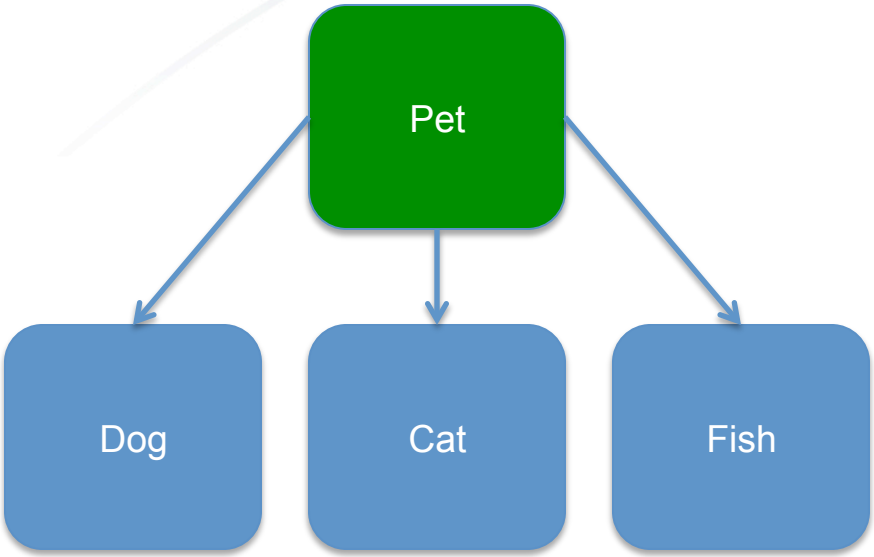
Python Classes – Inheritance Example

```
class Pet(object):

  def __init__(self,name, species):

    self.name=name
    self.species=species

  def getName(self): return self.name

  def getSpecies(self): return self.species

  def noise(self): return NotImplemented


class Dog(Pet):

  def __init__(self,name):
    super(Dog,self).__init__(name=name, species='dog')

  def noise(self): return 'bark'

class Cat(Pet):

  def __init__(self,name):
    super(Cat, self).__init__(name=name, species='cat')

  def noise(self): return 'meow'

class Fish(Pet):

  def __init__(self, name):
    super(Fish,self).__init__(name=name, species='fish')

  def noise(self): return None
```

- `Pet` is a base class (also called Super Class)
- `Dog`, `Cat`, and `Fish` inherit from `Pet`
  - All will have `getName` and `getSpecies` (code re-use!)
  - All must implement `noise`, otherwise an error is thrown if called.

What is *super*?

Python Classes – Pet Example

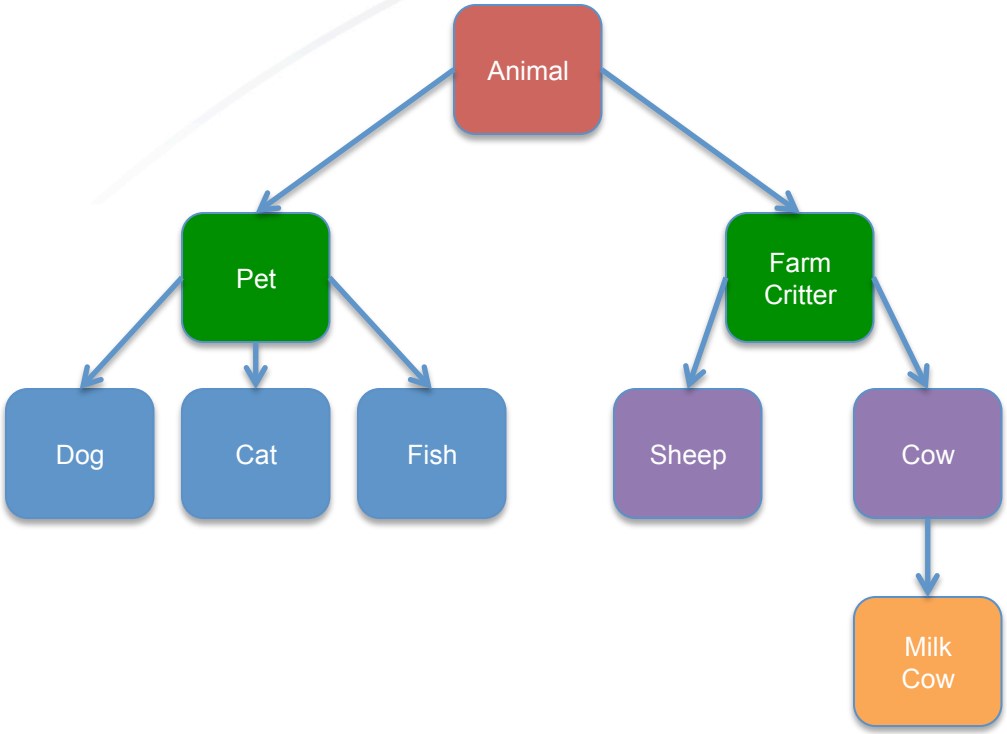


- In this simple inheritance structure, `super` calls the Pet `__init__` method to initialize an instance.
  - Useful to reduce code!
- What about more complicated inheritance patterns?

```
super(Dog,self).__init__(name=n,species=s)
```

Execute the Dog parent class (Pet) __init_ method.

Python Classes – Animal Example



- Create a base class `Animal` that requires a minimum of species and food.
  - The `noise` method in the previous `Pet` class will be defined here.
- Create two sub-classes, `Pet`, that includes a `name` attribute and `FarmCritter`, that includes a flag (`isEdible`) and a dollar value.
- MilkCow inherits from `Cow` and the `isEdible` flag is defined as `False`.

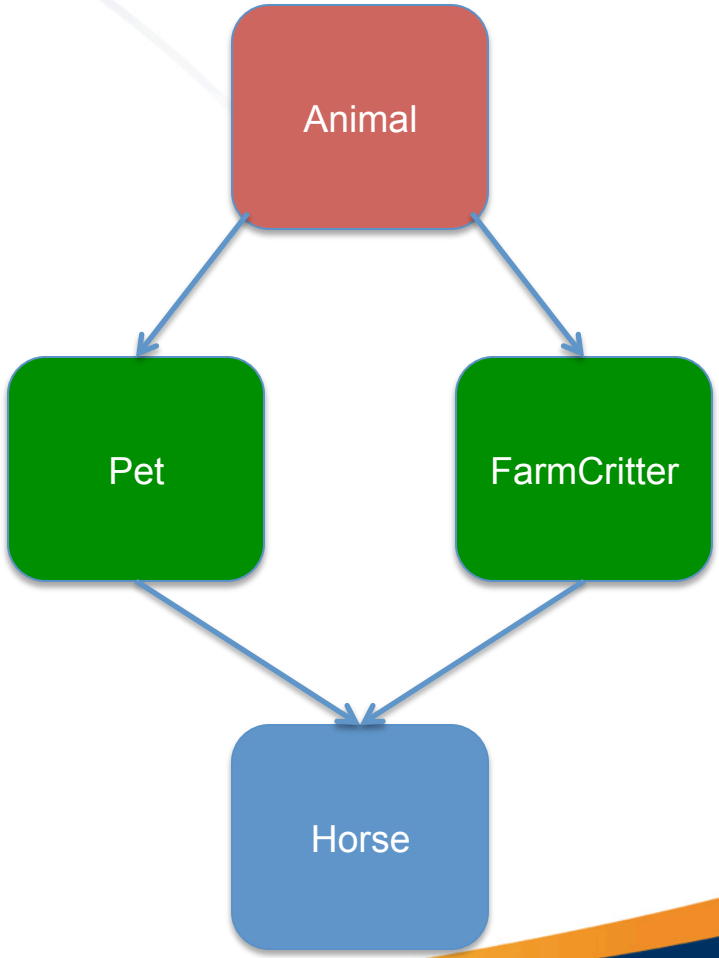
For `MilkCow`, what happens when *`super`* is called?

Python Classes – `zootopia.py`

- Simple script demonstrating `super`.
- Notice:
  - Multiple instances of `Dog` types (*`fido`* and *`buster`*) and since `Dog` is mutable an attribute like *`food`* can be changed after declaration. (See Buster's change in food preference)
  - Notice the print statements for `Cow` and `MilkCow` when instances for these classes are created.
  - One super statement ensures that all the __init__ are called in sequence.

Python Classes – Building a Horse

- What if we want to create a `Horse` class that inherits from `Pet` and `FarmCritter`?
- This creates what is called a diamond pattern of inheritance and `super` can not resolve which `__init__` to call once past `Animal`.
- Requires brute force to ensure the initialization in the order I want. See the script.
- Which `__str__` method is used? Both define `__str__`.
  - Answer: `FarmCritter`, because it is listed first in the inheritance declaration!



Python - Inheritance

As `Horse` class demonstrates, this can become very complicated. So why would we consider doing this?

Common interfaces to methods, but implementation can be altered for different instances. Here each animal has a unique definition for `noise`, but identical interface to make 'noise'

```
animals=[Cat('Fluffy'), Fish('Sharkbait'), Cow(60.00)]

for a in animals:
  print '%s says %s'%(a.species,a.noise())
```

Python - Inheritance

Easy to add a new type of Animal. Less lines of code reduces maintenance.

```
Class Chicken(FarmCritter):
  def __init__(self):
    super(Chicken,self).__init__(value=0.10,isEdible=True)
```

Methods that require Animal types do not need updating if a new sub-class of Animal is added.

```
def feedTheAnimals(*args):

  for a in args:
     feedFoodToAnimal(a.food)
```

Python – OO vs. Functional

Object Oriented Programming

- Organize around abstract things that own data and tasks performed on that data.
- Methods and data are co-located.
- More common methods (behavior) should be defined in base classes to reduce code.
  - Be prepared to change base class

Functional Programming

- Organize around an algorithm that requires input data.
- Procedure definitions and the data are not co-located.
- Different behavior requires different data and usually a different procedure.

Python - Importing Packages

How do use all the built-in or third party packages available in Python?

```
import sys
import os

from numpy.optimization import newton
from scipy.constants import N_A as AvoNum

path=os.path.join('usr','projects')

soln=newton(myfunc,x0)

num_particles=AtomicMass/AvoNum
```

`import packagename` adds *packagename* to the local scope.

Can add package names, classes or functions

The 'as' modifier is used to alias the name in the local scope

Environment variable PYTHONPATH controls where Python searches for packages.

Python – Defining Packages

- *module*: a file with class and function definitions
- *package*: a collection of modules
- Organize sub-packages into subdirectories that are also packages

```
material/
  __init__.py
  base.py
  solid.py
  gas.py
  eos/
    __init__.py
    base.py
    ideal.py
    sesame.py
```

Modules are `base.py`, `solid.py`, `gas.py`, etc.

Packages are `material` and `eos` is sub-package of `material`.

What's `__init__.py`?

Python Packages

material/__init__.py

```
from .solid import SolidMat
from .gas import GasMat
```

material/eos/__init__.py

```
from .sesame import SesameEos
from .ideal import IdealGasEos
```

The `__init__.py` is a special file that Python uses to determine what is visible through import of the package.

Allows the code that defines `SolidMat` to reside in a separate file.

`from .solid import SolidMat`

means:

"Look in the local (.) directory for the module file solid.py to find the code that defines class SolidMat."

```
from authority.material import SolidMat

from authority.material.eos import IdealGasEos
```

Python – Unit Tests

- One of the best features of the Python language is the `unittest` module.
- I could spend an entire hour on this.
- Unit tests exercise behavior of small units of code. As a project evolves, these tests will catch changes in behavior.
  - Remember OOP is designing abstract objects and their behavior.

Attempting to write OOP code without unit testing is ill-advised

- Resources
  - A good introduction to unit testing https://taco.visualstudio.com/en-us/docs/unit-test-01-primer/
  - Nose, an extended unit test Python framework http://nose.readthedocs.io/en/latest/index.html
- Write tests as you write code!

Python – Unit Test (Example)

```
from scipy.constants import N_A

class Particle(object):
  '''
  Base particle class that holds population value
  :param N float: Number of particles (default 0.0)
  '''

  def __init__(self,N=0.0):
    self._N=N

  @property
  def N(self): return self._N

  @N.setter
  def N(self,value):
    self._N=float(value)

class Nuclide(Particle):
  '''
  Base Nuclide class
  :param Z float: Atomic number
  :param A float: Atomic weight (AMU)
  :param N float: Number of particles (default 0.0)
  '''
  def __init__(self,Z,A,N=0.0):
    super(Nuclide,self).__init__(N)
    self.Z=Z
    self.A=A

  @property
  def mass(self):return (self.A*(self.N/N_A))

  @mass.setter
  def mass(self,value):
    self.N=((float(value)/self.A)*N_A)
```

```
import unittest

class TestNuclides(unittest.TestCase):

  def test_initial(self):
    from ..nuclides import Nuclide
    Dummy=Nuclide(Z=0.0,A=0.0)
    self.assertTrue(True)

  def test_N(self):
    from ..nuclides import Nuclide
    N0=1.0e+3
    C=Nuclide(Z=6.0,A=12.0107,N=N0)
    self.assertEqual(C.N,N0,'Failed to set initial
population value')
    self.assertEqual(C.Z,6.0,'Failed to set Z value')
    self.assertEqual(C.A,12.0107,'Failed to set A value')

  def test_mass(self):
    from scipy.constants import N_A
    from ..nuclides import Nuclide
    C=Nuclide(Z=6.0,A=12.0107)
    C.mass=12.0107
    self.assertEqual(C.N,N_A,'Failed to update the mass/N
correctly')
```

A simple test for import errors or typos

Check the attribute settings

Check N/mass updates

Python - Resources

- Use Google
  - Incredibly large and diverse community using Python in many scientific settings. If someone local can not answer your question, there is a very good chance someone on Reddit or StackOverflow has.
- Books
  - I do not recommend the O'Reilley books. Most of the information is found online for free. Although the pocket reference looks handy and cheap (<$10)
  - Python Scripting for Computational Science, H.P. Langtangen
- Web Pages
  - Well organized tutorial site: http://www.tutorialspoint.com/python/
  - The Newbie Python sub-reddit: http://www.reddit.com/r/learnpython
  - SciPy: http://www.reddit.com/r/scipy