References and an intro to Object-Oriented Programming (OOP)

Eric Franzosa, Ph.D. (franzosa@hsph.harvard.edu)

2019-12-16

Outline

- Final Project reminder
  - Due NEXT Monday (extended)
  - Bonus office hour this Weds (+ regular Friday hour)
- Grading reminder
  - Contact me if you are targetting Mar 2020 graduation
- Today's material
  - **Data vs. references**
  - **Intro to Object-Oriented Programming (OOP)**

Variables as references

- We've thought of variables as "buckets" for storing data
- This is a useful analogy, but as some of you are finding, it breaks down in certain cases

- In reality, data are stored in literal blocks of your computer memory
  - Represented as sequences of 0/1 values (bits) = binary code
- Variables are themselves data that point to (reference) other locations in memory

In [ ]:

```
# this code prints the memory address of the string data argument
hex( id( "Hello, World!") )
```

- The problems with the bucket analogy are less obvious with strings, numbers, and booleans because their can't be changed in place (they are immutable).

In [ ]:
```
# here, <a> and <b> refer to the same piece of data
a = "Hello, World!"
b = a
print( "<a> =", a, "@", hex( id( a ) ) )
print( "<b> =", b, "@", hex( id( b ) ) )
```

In [ ]:
```
# b.upper( ) returns NEW data
b = b.upper( )
```

In [ ]:
```
# <b> now refers to the new data; <a> still refers to the original data
print( "<a> =", a, "@", hex( id( a ) ) )
print( "<b> =", b, "@", hex( id( b ) ) )
```

- Let's try something similar with a *mutable* piece of data, i.e. a `list`

In [ ]:

```
# here, <a> and <b> refer to the same piece of data
a = [1,2,3]
b = a
print( "<a> =", a, "@", hex( id( a ) ) )
print( "<b> =", b, "@", hex( id( b ) ) )
```

In [ ]:

```
# b.append( ) alters the underlying list IN PLACE
b.append( 4 )
```

In [ ]:

```
# <a> and <b> continue to refer to the same (now modified data); <a>'s meaning h
as changed!
print( "<a> =", a, "@", hex( id( a ) ) )
print( "<b> =", b, "@", hex( id( b ) ) )
```

The same concepts in cartoon form

references-cartoon.png

The `is` operator

- Because comparing memory addresses by eye is hard, Python includes a special operator (`is`) that tests if its operands (usually variables) are pointing at the same memory location / piece of data.

In [ ]:
```
a = [1,2,3]
b = [1,2,3]
c = a
```

In [ ]:
```
a == b # returns True because <a> and <b> have equivalent values
```

In [ ]:
```
a is b # returns False because <a> and <b> were defined separately
```

In [ ]:
```
c is a # returns True because <a> and <c> point to the same data in memory
```

A helper function for the next few slides

In [ ]:

```
def compare( a, b ):
    print( "arg1 represents:", a )
    print( "arg2 represents:", b )
    print( "args have" + (" THE SAME " if a == b else " DIFFERENT ") + "value
(s)" )
    print( "args have" + (" THE SAME " if a is b else " DIFFERENT ") + "memory l
ocation(es)" )
    return None
```

Use `.copy( )` to create a new copy of `list/dict` data

In [ ]:

```
# here, <a> and <b> refer to the same piece of data
a = [1,2,3]
b = a.copy( )
compare( a, b )
```

In [ ]:

```
# empty slicing also works (lists only)
c = a[:]
compare( a, c )
```

- If you have a complex data structure, e.g. a list of lists, use `copy.deepcopy( )` instead
  - `.copy( )` is "shallow" - it only copies the structure of the outer list
  - data referenced inside the list would still be copied as a reference

In [ ]:
```
a = [[1,2],[3,4]]
b = a.copy( )
```

In [ ]:
```
# the outer lists are different
compare( a, b )
```

In [ ]:
```
# but the inner lists point to the same data (copy was "shallow")
compare( a[0], b[0] )
```

In [ ]:
```
# changing inner <b> element changes same data underlying <a> inner element
b[0][0] = "Hello, World!"
print( a[0][0] )
```

- `copy.deepcopy( )` fixes this behavior

In [ ]:
```
from copy import deepcopy
a = [[1,2],[3,4]]
b = deepcopy( a )
```

In [ ]:
```
# now even the nested data is different
compare( a[0], b[0] )
```

In [ ]:
```
# <a>'s inner data not perturbed by changing <b>
b[0][0] = "Hello, World!"
print( a[0][0] )
```

Data are passed to functions by reference

- Which can result in functions changing mutable data unexpectedly when provided as an argument

In [ ]:
```
a = [1,2,3]
```

In [ ]:
```
def sum_squares( numbers ):
    for i in range( len( numbers ) ):
        numbers[i] = numbers[i] ** 2
    return sum( numbers )
```

In [ ]:
```
# sum_squares returns the expected sum...
sum_squares( a )
```

In [ ]:
```
# ... but it also updated <a> in the process! (Surprise?)
print( a )
```

What is Object-Oriented Programming (OOP)?

- A *style* of programming that bundles data with related methods
- These bundles are called *classes* (or *types*)
- Classes are templates for making *instances* of a particular kind of data object
    - e.g. `str`, `list`, and `numpy.ndarray` are classes
- OOP style asks data to perform actions, rather than applying transformations *to* data
    - e.g. `str.upper( )`, `list.sort( )`

In [ ]:

```
# the type( ) function tells us what type the given data belongs to
type( "Hello, World!" )
```

Key OOP ideas

- Classes are organized hierarchically as superclasses and subclasses
  - This allows us to define progressively more specific versions of objects
  - *Thing > Animal > Mammal > Cow*
  - *Thing > Animal > Mammal > Cat*

- Classes inherit the attributes and abilities of their parent classes (*inheritance*)
  - `Mammal` has a method `produce_milk`
  - Hence `Cow.produce_milk( )` works
  - Hence `Cat.produce_milk( )` works

- Different classes of object can respond to the same request in different ways
  - Referred to as *polymorphism*
  - `Cow.speak( )` returns "moo"
  - `Cat.speak( )` returns "meow"

Defining our own classes of object

- Not every program/project needs new classes of object
  - In my experience, *much* less common than new functions, for example
- They become handy when built-in data types (e.g. `list` and `dict`) come up short
- Let's look at an example where this is the case

Modeling doors

- A door is an object with at least two obvious attributes:
    1. Some sort of unique identifier (e.g. a door number)
    2. A closed/open status

In [ ]:

```python
# Python lets us store misc. attributes as lists; is a list a good door?
door1 = [101, True]
door2 = [102, False]
```

In [ ]:

```python
# dictionaries let us name the attributes, which is a bit better
door1 = {"number": 101, "is_open":True}
door2 = {"number": 102, "is_open":False}
```

In [ ]:
```
# we can define transformations for a door
def open_door( door ):
    door["is_open"] = True
```

In [ ]:
```
door2
```

In [ ]:
```
open_door( door2 )
door2
```

- Later I realize that doors can have another status: locked/unlocked

In [ ]:
```
# I start adding this field to my door dictionaries from now on
door3 = {"number": 103, "is_open":False, "is_locked":True}
```

In [ ]:
```
# I also need to update the opening function
def open_locking_door( door ):
    if not door["is_locked"]: # <--
        door["is_open"] = True
```

In [ ]:
```
door3
```

In [ ]:
```
open_locking_door( door3 )
door3
```

In [ ]:

```
# the new opening function won't work on our earlier-defined doors
open_locking_door( door2 )
```

Issues with the above approach

- I'm relying on my memory to track the dictionaries we created as "doors"
- There is nothing enforcing the requirements to be a "door"
    - is `{"number":104, "is_locked":True}` a "door"?
- There is nothing tying the door transformations we wrote to the door data
- There is nothing tying the locked door to the more generic door

Defining a `Door` object

In [ ]:

```
class Door:
    def __init__( self, number ):
        self.number = number
        self.is_open = False
```

- `class` is a Python keyword for defining a new type of object with a block of code
- The block encapsulates relevant functions (*methods*) and data (*attributes*)
- The `__init__` method defines what happens when we make a new instance of the object
  - Here, set a number (passed as an argument) as the Door's number
  - Also, create an attribute `is_open` set to False
- `self` is used to refer to the object itself in methods (more in a bit)

- Calling a `Door` like a function runs its `__init__` method and returns a new door
  - Python's `__init__` is called a *constructor* in other languages

In [ ]:

```
# make a new Door numbered 101
door1 = Door( 101 )
```

In [ ]:

```
# Python sees this door as a new kind of object
print( door1 )
```

In [ ]:

```
# access Door attributes with <.> syntax
door1.number
```

In [ ]:

```
# note that <is_open> we defined as False by default
door1.is_open
```

- We can associate other Door-related methods with the `Door` class

In [ ]:
```
class Door:

    def __init__( self, number ):
        self.number = number
        self.is_open = False

    def open( self ):
        self.is_open = True

    def check_status( self ):
        print( "I'm open" if self.is_open else "I'm closed" )
```

- The method call `door1.check_status( )` behaves like a function call `check_status( door1 )`
- The `self` argument of `check_status` is what allows this to work
  - `door1.check_status( )` means "call `check_status` with `door1` as the first argument"
  - Hence `self` is always present as the first argument of a method

In [ ]:

```
door1 = Door( 101 )
# call Door methods using <.> syntax
door1.check_status( )
```

In [ ]:

```
door1.open( )
door1.check_status( )
```

In [ ]:
```python
# let's make some more Doors
door2 = Door( 102 )
door3 = Door( 103 )
```

In [ ]:
```python
# we can interact with them efficiently
for d in [door1, door2, door3]:
    d.check_status( )
```

In [ ]:

```
# oops, I accidentally repeated a door number
door4 = Door( 103 )
```

In [ ]:

```
# door3 and door4 are different, even though their attributes are all the same
door3 == door4
```

In [ ]:

```
# and verified by their memory addresses
print( "<door3> is located @", hex( id( door3 ) ) )
print( "<door4> is located @", hex( id( door4 ) ) )
```

In [ ]:

```
# compare with
door3 = {"number": 103, "is_open":True}
door4 = {"number": 103, "is_open":True}
door3 == door4
```

The power of `Door` (i.e. OOP)

- We don't have to rely on our memory for definition
  - Need a door? Call `Door`
- Can have required (e.g. `number`) and default (e.g. `is_open`) attributes
- Relevant methods are associated with the object (e.g. `open`)
- Object is distinct from the sum of the data it contains
- *Next up: We can easily make other types of doors*

Defining a `SecureDoor` object

In [ ]:
```
class SecureDoor( Door ):

    def __init__( self, number ):
        # use the Door constructor to start setup of this door
        Door.__init__( self, number )
        # finish by adding a new attribute: <is_locked>
        self.is_locked = True

    # REDEFINE open( ) to check <is_locked>
    def open( self ):
        if not self.is_locked:
            self.is_open = True
```

- `class SecureDoor( Door )` says `SecureDoor` is a type of `Door`
- By default, `SecureDoor` inherits all the methods and attributes of `Door`
- We've added a new attribute to the `__init__`: `is_locked`
- We've reworked open to check `is_locked`
- We didn't redefine `check_status`

In [ ]:
```
# let's make a secure door
sec_door = SecureDoor( 105 )
```

In [ ]:
```
# SecureDoor inherits the <check_status> method from Door
sec_door.check_status( )
```

In [ ]:
```
# But its <open> method works differently
sec_door.open( )
sec_door.check_status( )
```

Because we have implemented an `open` method in all doors, we can still do intuitive things like:

In [ ]:
```
# polymorphism: <open> works differently on different doors
for d in [door1, door2, sec_door]:
    d.open( )
    d.check_status( )
```

Practical example: Defining an `Interval` class

- Could represent a span of years, e.g. 1983-2018
- Could represent a span of genome coordinates, e.g. 1,383,452 to 1,384,591

In [ ]:
```
# an interval is defined by a start and end position
class Interval( ):
    def __init__( self, start, end ):
        self.start = start
        self.end = end
```

In [ ]:
```
ival1 = Interval( 1983, 2018 )
```

In [ ]:
```
print( ival1 )
```

In [ ]:
```
ival1.start, ival1.end
```

- A lot of Python polymorphism comes from implementing special object methods flanked by __s
- For example, implement `__repr__` to define interaction with the `print` function
- This is also the method that is called if we evaluate a piece of data on its own line in a Jupyter Notebook

In [ ]:
```
class Interval( ):

    def __init__( self, start, end ):
        self.start = start
        self.end = end

    def __repr__( self ):
        return "Interval( " + str( self.start ) + " -> " + str( self.end ) + " )"
```

In [ ]:
```
ival1 = Interval( 1983, 2018 )
```

In [ ]:
```
print( ival1 )
```

In [ ]:
```
ival1
```

- Implement `__len__` to determine interaction with the `len` function

In [ ]:
```
class Interval( ):

    def __init__( self, start, end ):
        self.start = start
        self.end = end

    def __repr__( self ):
        return "Interval( " + str( self.start ) + " -> " + str( self.end ) + " )"

    def __len__( self ):
        return self.end - self.start
```

In [ ]:
```
ival1 = Interval( 1983, 2018 )
```

In [ ]:
```
len( ival1 )
```

- The length of a discrete interval is different from that of a continuous interval
    - *We must include the end point as a unit of distance*
- For example, the interval from 2 to 4 in 1->2->3->4->5 contains 3 numbers
- This is a great use-case for subclassing/polymorphism

In [ ]:

```
class DiscreteInterval( Interval ):

    # Note: no <__init__>, we can just inherit the one from <Interval>

    def __len__( self ):
        return self.end - self.start + 1
```

In [ ]:

```
ival1 = DiscreteInterval( 2, 4 )
```

In [ ]:

```
len( ival1 )
```

- Let's extend `Interval` to make a better interval with an extra method
- Specifically, one that will test if the interval contains a particular value

In [ ]:
```
class BetterInterval( Interval ):

    def contains( self, value ):
        """ returns True if <value> in the interval """
        return self.start < value < self.end
```

In [ ]:
```
ival1 = BetterInterval( 1983, 2018 )
```

In [ ]:
```
ival1.contains( 1776 )
```

In [ ]:
```
ival1.contains( 1995 )
```

- Let's extend `Interval` (again) to make a better interval with an extra method
- This time, let's define an interval that can test if it overlaps with some other interval
- HINT: two intervals overlap if the LARGER START is smaller than the SMALLER END

In [ ]:
```
class BetterInterval( Interval ):

    def overlaps( self, ival2 ):
        """ return True if this interval overlaps ival2 """
        return max( self.start, ival2.start ) < min( self.end, ival2.end )
```

In [ ]:
```
ival1 = BetterInterval( 1983, 2018 )
# note that second interval doesn't have to be a <BetterInterval>
ival2 = Interval( 1969, 1995 )
ival3 = Interval( 1969, 1974 )
```

In [ ]:
```
ival1.overlaps( ival2 )
```

In [ ]:
```
ival1.overlaps( ival3 )
```

- Let's make a final interval that will merge two overlapping intervals as a new interval

In [ ]:
```
class BestInterval( BetterInterval ):

    def merge( self, ival2 ):
        ret = None
        if self.overlaps( ival2 ):
            min_start = min( self.start, ival2.start )
            max_end = max( self.end, ival2.end )
            ret = BestInterval( min_start, max_end )
        return ret
```

In [ ]:
```
ival1 = BestInterval( 1983, 2018 )
ival2 = Interval( 1969, 1995 )
ival3 = Interval( 1969, 1974 )
```

In [ ]:
```
print( ival1.merge( ival2 ) )
```

In [ ]:
```
print( ival1.merge( ival3 ) )
```

- If we define our `merge` function as `__add__` instead, then we can use the addition operator (+) to merge intervals
- This is how + can add numbers but concatenate strings in Python: Polymorphism!

In [ ]:
```
class BestInterval( BetterInterval ):
    def __add__( self, ival2 ):
        ret = None
        if self.overlaps( ival2 ):
            min_start = min( self.start, ival2.start )
            max_end = max( self.end, ival2.end )
            ret = BestInterval( min_start, max_end )
        return ret
```

In [ ]:
```
ival1 = BestInterval( 1983, 2018 )
ival2 = Interval( 1969, 1995 )
ival3 = Interval( 1969, 1974 )
```

In [ ]:
```
ival1 + ival2
```

Practical example: Defining a `SimpleCounter` class

- For counting the elements of iterable objects
- A task that came up on numerous homeworks

In [ ]:

```
class SimpleCounter( ):

    def __init__( self ):
        self.counts = {}

    def update( self, iterable ):
        for thing in iterable:
            if thing not in self.counts:
                self.counts[thing] = 0
            self.counts[thing] = self.counts[thing] + 1

    def __repr__( self ):
        return str( self.counts )
```

In [ ]:

```
sc = SimpleCounter( )
sc.update( "bananarama" )
print( sc )
```

- Let's subclass `SimpleCounter` to make something a bit more aesthetically pleasing
- We'll redefine `__repr__`, but `__init__` and `update` don't need to change

In [ ]:

```
class PrettyCounter( SimpleCounter ):
    def __repr__( self ):
        ret = []
        for item, count in self.counts.items( ):
            ending = "s" if count > 1 else ""
            ret.append( "I found '{}' {:>2} time{}".format( item, count, ending ) )
        return "\n".join( ret )
```

In [ ]:

```
pc = PrettyCounter( )
pc.update( "bananarama" )
pc.update( "ana, my nana, ate a banana" )
print( pc )
```

- As you may have discovered, there's a similar `Counter` in the `collections` module:

In [ ]:
```
from collections import Counter
cc = Counter( )
cc.update( "bananarama" )
print( cc )
```

Nothing magic about the "official" `Counter` - it works just like ours!

Practical example: `Tree` data

- A tree is a general data structure in which items (called nodes) are arranged hierarchically
- The tree begins at a `root` node
- All other nodes have exactly one `parent`
- A node can therefore have `0` or more `children`

In [ ]:

```
# The class to represent a <Node> is not too complicated

class Node( ):

    def __init__( self, name ):
        self.name = name
        self.parent = None
        self.children = []

    def __repr__( self ):
        return self.name
```

In [ ]:

```python
# The class to represent a <Tree> is more involved (it does most of the work)

class Tree( ):

    def __init__( self, ):
        """ a dictionary to map node names to nodes in the tree """
        self.nodes = {}

    def get_node( self, name ):
        """ fetch an existing node by name, or create it if new """
        if name not in self.nodes:
            self.nodes[name] = Node( name )
        return self.nodes[name]

    def populate( self, relationships ):
        """ add parent/child relationships to the tree """
        for parent, child in relationships:
            pnode = self.get_node( parent )
            cnode = self.get_node( child )
            cnode.parent = pnode
            pnode.children.append( cnode )
```

In [ ]:

```
relationships = [
    ["thing", "vehicle"],
    ["thing", "animal"],
    ["vehicle", "plane"],
    ["vehicle", "train"],
    ["vehicle", "automobile"],
    ["animal", "mammal"],
    ["mammal", "cat"],
    ["mammal", "cow"],
]
```

In [ ]:

```
my_tree = Tree( )
my_tree.populate( relationships )
```

In [ ]:

```
for name, node in my_tree.nodes.items( ):
    print( node )
    print( "  parent   :", node.parent )
    print( "  children :", node.children )
```

Challenges

- Add a method to `Tree` called `get_root` that will find and return the tree's root node (hint: in a properly defined tree, the root is the only node that doesn't have a parent).
- Add a method to `Tree` called `get_leaves` that will find and return the tree's leaf nodes (hint: a leaf is a node that doesn't have any children of its own).
- (*Harder*) Add a method to `Tree` called `get_lineage`. This function should take the name of a node as an argument and return the path from the root of the tree to that node. For example `my_tree.get_lineage( 'cow' )` should return `['thing', 'animal', 'mammal', 'cow']` based on the data above.