W18 Reading Unit 8

July 14, 2016

1 Object-Oriented Programming

You don't always need clsses. For smll software projects, or for a script to run a one-time analysis of a data set, it is okay to program in a script style.

But as they grow larger -> onject oriented programming

```
Codig becomes a process of desingning types and the interfaces that governtheir interaction
```

When program gets bigger and classes might be similar, we need **inheritance** (a relationship between classes)

when we create a class, we can also specify a parent clas. a child class inherits the attributes of its parent. The child class can also define new attributes, or overwrite those of its parent.

One class can have multiple children. in fact a class can have multiple parents in Python too.

Python has an entire hierarchy of classes and we add on to that hierarchy when we make our own classes. at the very top of the hierachy is the class "object" everthing we create actually inherie from this object class.

suppose we want a variable, self.x or a method, self.x() inside an object. How does Python find the attribute?

1. Look in the instance for attribute
2. If not in the instance, look to the object's class for attribute
3. if not in the object's class, look up the hierarchy of that class for attribute.
4. if you hit object, then the attribute does not exist

Benefits of Inheritance

1. Inheritancs allows subclasses to reuse coode found in parent class
2. Instead of writing a class from scratch, one can try to specialize an existing class by extending it.
3. Parent clss can define an interface that allows many subclasses to interact with a program
4. Inheritance allows a programmer to organize related objects.

1.1 Class inheritance

1.1.1 simulations

Stochastic Process : a single varibale changes over time

(we are looking at discrete time) stochastic process x_1 x_2 $\ldots$

e.g.

1. stock price over time
2. measurements of solar radiation for each day
3. average planetary surface temperature

```
In [39]: class Process:
             """Represetation of Stochastic Process"""

             def __init__(self, start_value = 0):
                 self.value = start_value # define an attribute

             def time_step(self):
                 raise NotImplementedError()
                 #pass #pass basiclly means does nothing
```

```
In [40]: p1 = Process() # define a variable p1
         p1.time_step() # call the time_step() method
```

```
        ---------------------------------------------------------------------------

        NotImplementedError                       Traceback (most recent call last)

        <ipython-input-40-96d0c33b6112> in <module>()
          1 p1 = Process() # define a variable p1
    ----> 2 p1.time_step() # call the time_step() method


        <ipython-input-39-53098bdcd76b> in time_step(self)
          5
          6     def time_step(self):
    ----> 7         raise NotImplementedError()
          8         #pass #pass basiclly means does nothing
          9


        NotImplementedError:
```

another way is to use **pass** which basically does nothing

```
In [159]: class Process:
              """Represetation of Stochastic Process"""
              def __init__(self, start_value = 0):
                  self.value = start_value # define an attribute

              def time_step(self):
                  pass #pass basiclly means does nothing
```

```
In [52]: p1 = Process() # create a variable p1
         print("The default p1.value is: ", p1.value)
         p1.time_step() # call the time_step() method, which does nothing right now

         p2 = Process(10) # create a variable p2
         print("We assigned a value 10 when we create p2, and"
               " p2.value is: ", p2.value, "instead of the default 0")
```

```
The default p1.value is:  0
We assigned a value 10 when we create p2, and p2.value is:  10 instead of the default 0
```

let's go ahead and create a sub class

```
In [160]: class BoundeLinearProcess(Process):
              """A stochastic process that develops linearly. Increases
              by velocity in every time period, but is bounded between 0 and 1."""

              def __init__(self,start_value = 0,velocity = 0):
                  # this interesting super function actually returns to the super
                  # class which in this case the Process, the parent class.
                  # by adding __init__() we go to the Process init function
                  super().__init__(start_value)
                  self.velocity = velocity

              def time_step(self):
                  # this function will first add the velocity to its initial value
                  # which assigned to self.value.
                  # and than, it will check if the updated self.value is less than 0
                  # if it is < 0, it will reflect the value and the velocity
                  # if self.value is actually greater than 1, it will do the thing
                  # defined in the if statement
                  self.value += self.velocity
                  if self.value < 0:
                      self.value = -self.value
                      self.velocity = -self.velocity
                  if self.value > 1:
                      self.value = 1 - (self.value -1)
                      self.velocity = -self.velocity
                  # this go to the Process super eventhough now super does nothing
                  super().time_step
```

```
In [61]: p1 = BoundeLinearProcess(0, .3)
         # we creat a object p1 with start_value = 0 and velocity = 0.3
         # notice we can do this because when we def the  __init__ function in
         # BoundeLinearProcess class, there are two initial values: start_value
         # and velocity
         for i in range(4):
             # than we call time_step 4 times
             p1.time_step()
             print("Current Process value: ", p1.value)
```

```
Current Process value:  0.3
Current Process value:  0.6
Current Process value:  0.8999999999999999
Current Process value:  0.8
```

There is a good way to check your instance without doing this print statement.
The use of _ str _(self)
The use of _ repr _(self)

```
In [161]: class Process:
              """Represetation of Stochastic Process"""
              def __init__(self, start_value = 0):
                  self.value = start_value # define an attribute

              def time_step(self):
                  pass #pass basiclly means does nothing

              def __str__(self):
                  # this is a "magic method"
                  return "Process with current value" + str(self.value)
          # or we can add the floowing if str does not work
              #def __repr__(self):
              #    # this is another "magic method"
              #    return __str__(self)
```

```
In [71]: class BoundeLinearProcess(Process):
             """A stochastic process that develops linearly. Increases
             by velocity in every time period, but is bounded between 0 and 1."""

             def __init__(self,start_value = 0,velocity = 0):
                 super().__init__(start_value)
                 self.velocity = velocity

             def time_step(self):
                 self.value += self.velocity
                 if self.value < 0:
                     self.value = -self.value
                     self.velocity = -self.velocity
                 if self.value > 1:
                     self.value = 1 - (self.value -1)
                     self.velocity = -self.velocity
                 super().time_step
```

```
In [141]: p1 = BoundeLinearProcess(0, .3)
          for i in range(4):
              p1.time_step()
              print(p1)
```

```
*
            *
                  *
                *
```

Now lets check how to override _ str _(self)

```
In [77]: class BoundeLinearProcess(Process):
             """A stochastic process that develops linearly. Increases
             by velocity in every time period, but is bounded between 0 and 1."""

             def __init__(self,start_value = 0,velocity = 0):
                 super().__init__(start_value)
                 self.velocity = velocity

             def time_step(self):
                 self.value += self.velocity
                 if self.value < 0:
                     self.value = -self.value
                     self.velocity = -self.velocity
                 if self.value > 1:
                     self.value = 1 - (self.value -1)
                     self.velocity = -self.velocity
                 super().time_step

             def __str__(self):
                 return " " * int(self.value*20) + "*"
```

```
In [80]: p1 = BoundeLinearProcess(0, .1)
         for i in range(20):
             p1.time_step()
             print(p1)
```

```
*
    *
      *
        *
          *
            *
              *
               *
                  *
                   *
                  *
                *
              *
            *
          *
        *
      *
    *
  *
*
```

1.2 More Class Inheritance

```
In [142]: class Process:
              """Represetation of Stochastic Process"""
              def __init__(self, start_value = 0):
                  self.value = start_value # define an attribute

              def time_step(self):
                  pass

              def __str__(self):
                  return "Process with current value" + str(self.value)

              def simulate(self,steps = 20):
                  for i in range(steps):
                      print(self)
                      self.time_step()
```

```
In [143]: class BoundeLinearProcess(Process):
              """A stochastic process that develops linearly. Increases
              by velocity in every time period, but is bounded between 0 and 1."""

              def __init__(self,start_value = 0,velocity = 0):
                  super().__init__(start_value)
                  self.velocity = velocity

              def time_step(self):
                  self.value += self.velocity
                  if self.value < 0:
                      self.value = -self.value
                      self.velocity = -self.velocity
                  if self.value > 1:
                      self.value = 1 - (self.value -1)
                      self.velocity = -self.velocity
                  super().time_step

              def __str__(self):
                  return " " * int(self.value*20) + "*"
```

Autoregressive Process of order 1.
AR(1)
$x_t = \alpha x_{t-1} + w_t$

```
In [144]: import numpy as np

          class ARProcess(Process):   # create the ARProcess under the parent Process
              def __init__(self, alpha = 0.5, sigma = 1, start_value = 0):
                  super().__init__(start_value)
                  self.alpha = alpha
                  self.sigma = sigma

              def time_step(self):
                  self.value = self.alpha * self.value \
                               + np.random.normal(scale = self.sigma)
                  super().time_step()

              def __str__(self):
                  if self.value < 0:
                      s = " " *int( 5 * (self.value + 3)) + "*"  + " " * int(-self.value * 5) + "|"
                  elif self.value == 0:
                      s = " " * 15 + "*"
                  else:
                      s = " " * 15 + "|" + " " * int(5*self.value) + "*"
                  return s
```

```
In [145]: p1= BoundeLinearProcess(0,0.1)
          p2 = ARProcess(alpha = .9)
```

```
In [146]: p1.simulate()
          p2.simulate()
```

```
*
  *
    *
      *
        *
          *
            *
```

```
              *
               *
                  *
                   *
                  *
                *
              *
            *
          *
        *
      *
    *
  *
               *
               |     *
               |*
               |    *
               |         *
               |                       *
               |              *
               |              *
               |   *
               |   *
            *  |
               |*
               |       *
               | *
            *  |
        *      |
          *    |
             * |
        *      |
   *           |
```

Random Walk

$x_t = x_{t-1} + w_t$

Notice this is very similar to the previous one the only difference is that the alpha is now equals to 1

```
In [156]: # We can see that this RandomWalk is so easy to create !!!!!
          class RandomWalk(ARProcess):
              def __init__(self, sigma = 1):
                  super().__init__(alpha = 1, sigma = sigma)
```

```
In [158]: p3 = RandomWalk()
          p3.simulate()
```

```
*
            *  |
               |       *
               |            *
               |          *
               |     *
               |    *
               |       *
               |      *
|      *
|  *
|       *
|    *
|   *
|     *
|        *
|               *
|                    *
|                              *
|                                 *
```

1.3 Object-Oriented Programming

It is a way of conceptualizing an entire program as a set of objects that interact with each other - a focus placed on objects instead of tasks. All of the tasks that have to be done in a program are encoded in the behavior of objects

- instead of a method to scramble text, start thinking about a text scrambler class.
  - what attributes would makes sense for this class?
  - How should it interact with other objects in your program?
- A problem must be divided into a set of component objects that have behaviors and pass messages to each other.

1.3.1 Principles of OOP

- **Encapsulation**: we can break a problem down into different layers
  - there can be calsses that do low-level tasks and present a simple abstraction to other classes.
  - other classes can relate to high-level tasks and use the functionality in lower-level classes.
- **Modularity**: we can easily switch out one class for another to introduce different functionality and envolve a program.
- **Inheritance**: classes can take on the attributrs of parent classes. We can easily organize related classes and understand their funtionality
- **Polymorphism**: "polymorphism describes a pattern in object oriented programming in which classes have different functionality while sharing a common interface."
  - We can write diffrent classes that define the same attributes.
    - They have different behaviours on the inside but are used in the same way.
    - They share the same interface
  - You can write code that can work with these different types of objects, and it doesn't have to know which exact type it has
  - e.g.
    - say we have a function def is_passing(student):

    return student.grade >70.

    - pass in an object of type UndergradStudent or one of type GradStudent.
      - As long as they both define a grade attribute, out function doesn't have to know what type of object it has.

- we could even pass in an object Gasoline as long as it has the attribute grade. **Python does not check the types of objects at all!**At runtime, you just need to check that the objects you use have the attributes needed. This is something called **Duck typing**.

Duck Typing helps speed up development, but there's potential for more errors at runtime!

1.4 Using Polymorphism

```
In [2]: import numpy as np

        class Process:
            def __init__(self, start_value = 0):
                self.value = start_value # define an attribute
                self.history = []

            def time_step(self):
                self.history.append(self.value)

            def __str__(self):
                return "Process with current value" + str(self.value)

            def simulate(self,steps = 20):
                for i in range(steps):
                    self.time_step()

        class BoundeLinearProcess(Process):
            def __init__(self,start_value = 0,velocity = 0):
                super().__init__(start_value)
                self.velocity = velocity

            def time_step(self):
                self.value += self.velocity
                if self.value < 0:
                    self.value = -self.value
                    self.velocity = -self.velocity
                if self.value > 1:
                    self.value = 1 - (self.value -1)
                    self.velocity = -self.velocity
                super().time_step

            def __str__(self):
                return " " * int(self.value*20) + "*"

        class ARProcess(Process):
            def __init__(self, alpha = 0.5, sigma = 1, start_value = 0):
                super().__init__(start_value)
                self.alpha = alpha
                self.sigma = sigma

            def time_step(self):
                self.value = self.alpha * self.value \
                             + np.random.normal(scale = self.sigma)
                super().time_step()

            def __str__(self):
                if self.value < 0:
                    s = " " *int( 5 * (self.value + 3)) + "*"  + " " * int(-self.value * 5) + "|"
                elif self.value == 0:
                    s = " " * 15 + "*"
                else:
                    s = " " * 15 + "|" + " " * int(5*self.value) + "*"
                return s

        class RandomWalk(ARProcess):
            def __init__(self, sigma = 0.5):
                super().__init__(alpha = 1, sigma = sigma)
```

```
In [3]: class ProcessPlotter:
            """An object to display the history of a process"""
            def __init__(self, process = None):
                self.process = process

            def plot(self):
                pass
```

```
In [4]: class TextProcessPlotter(ProcessPlotter):
            def plot(self):
                upper = max(self.process.history)
                lower = min(self. process.history)
                if upper == lower:
                    upper += 1
                for val in self.process.history:
                    print(" " * int(20 * (val-lower)/(upper - lower)) + "*")
```

```
In [5]: p1 = ARProcess(alpha = 0.9)
        plotter1 = TextProcessPlotter(p1)
        p1.simulate(10)
        plotter1.plot()
```

```
*
        *
             *
        *
                 *
                    *
               *
        *
  *
       *
```

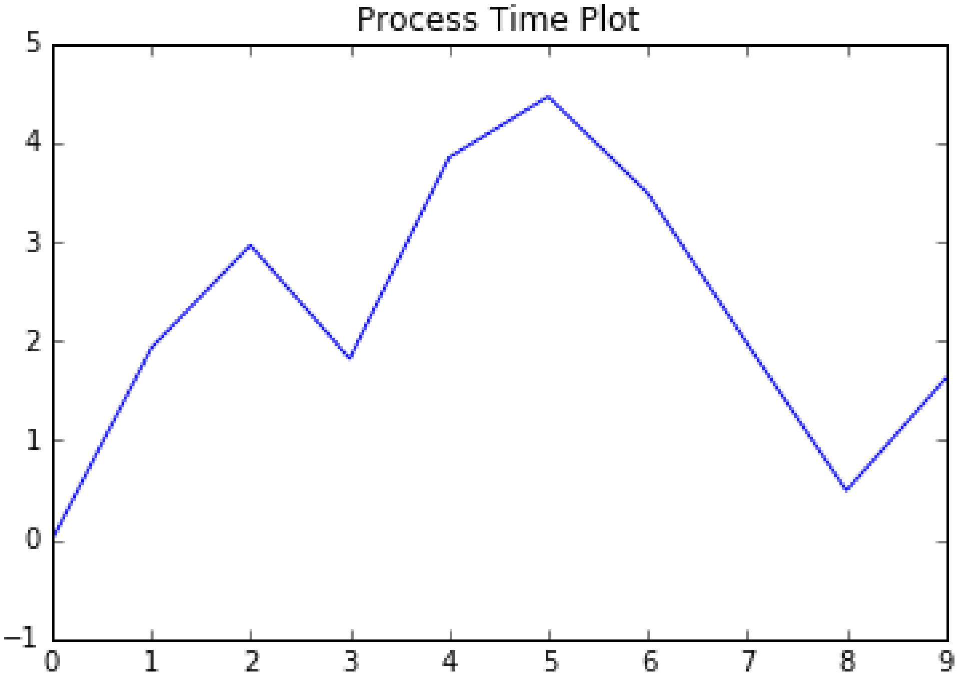
```
In [7]: p2 = RandomWalk()
        plotter2 = TextProcessPlotter(p2)
        p2.simulate()
        plotter2.plot()
```

```
*
                    *
           *
              *
```

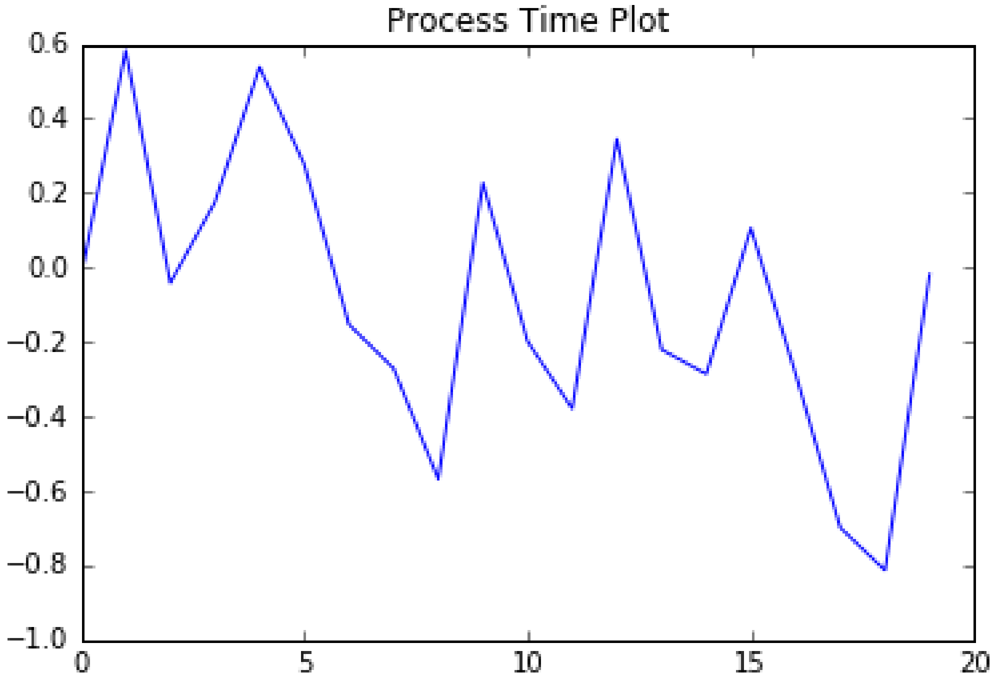
```
                   *
               *
         *
       *
   *
              *
        *
      *
                *
        *
       *
             *
       *
 *
*
           *
```

```
In [9]: import matplotlib
        %matplotlib inline
        class GraphicProcessPlotter(ProcessPlotter):
            def plot(self):
                matplotlib.pyplot.plot(self.process.history)
                matplotlib.pyplot.title("Process Time Plot")
```

```
In [11]: plotter3 = GraphicProcessPlotter(p1)
         plotter3.plot()
```



```
In [13]: plotter4 = GraphicProcessPlotter(p2)
         plotter4.plot()
```



1.5 Magic Methods

The magic method in python is denoted with the double underscore

```
In [23]: class Card:
             def __init__(self,value,suit):
                 self.value = value
                 self.suit = suit
```

```
In [14]: Card(5,"Spades") == Card(6,"Diamonds")
```

```
Out[14]: False
```

```
In [22]: print(Card(5,"Spades") == Card(5,"Spades"))
         print()
         print("The reason why the previous comparison is not equal"
               " is because if you do the following you can se that: ")
         print("Card(5,\"Spades\") is: ", Card(5,"Spades"))
         print("Card(6,\"Spades\") is: ", Card(6,"Spades"))
         print()
         print("They are just memory addreses!")
```

```
False

The reason why the previous comparison is not equal is because if you do the following you can se that:
Card(5,"Spades") is:  <__main__.Card object at 0x000001E952C1A048>
Card(6,"Spades") is:  <__main__.Card object at 0x000001E952C1A0B8>

They are just memory addreses!
```

```
In [29]: class Card:
             def __init__(self,value,suit):
                 self.value = value
                 self.suit = suit

             def __eq__(self,other):
                 if self.value == other.value:
                     return True
                 else:
                     return False
```

```
In [30]: print(Card(5,"Spades") == Card(5,"Spades"))
         print()
         print("Now if we use the magic method __eq__, it is true now")
         print("The __eq__ checks the content not the memory address")
```

```
True

Now if we use the magic method __eq__, it is true now
The __eq__ checks the content not the memory address
```

```
In [31]: # Let’s do some more!
         class Card:
             def __init__(self,value,suit):
                 self.value = value
                 self.suit = suit

             def __eq__(self,other):
                 if self.value == other.value:
                     return True
                 else:
                     return False
             def __lt__(self,other):
                 if self.value < other.value:
                     return True
                 else:
                     return False
             def __gt__(self,other):
                 if self.value > other.value:
                     return True
                 else:
                     return False
```

```
In [33]: cards = []
         for suit in [’Hearts’,’Spades’,’Diamonds’,’Clubs’]:
             for value in range(0,13):
                 cards.append(Card(value,suit))
```

```
In [36]: # Let’s see the first 5
         # The result obviously does not make any use to us
         cards[:5]
```

```
Out[36]: [<__main__.Card at 0x1e952c02b70>,
          <__main__.Card at 0x1e952c02a58>,
          <__main__.Card at 0x1e952c025c0>,
          <__main__.Card at 0x1e952c029e8>,
          <__main__.Card at 0x1e952c02f28>]
```

```
In [38]: cards[0] < cards[12]

Out[38]: True

In [40]: # Let’s do some more!
         class Card:
             def __init__(self,value,suit):
                 self.value = value
                 self.suit = suit

             def __eq__(self,other):
                 if self.value == other.value:
                     return True
                 else:
                     return False
             def __lt__(self,other):
                 if self.value < other.value:
                     return True
                 else:
                     return False
             def __gt__(self,other):
                 if self.value > other.value:
                     return True
                 else:
                     return False
             def __repr__(self):
                 return"%i of %s" % (self.value,self.suit)

In [42]: #now, it looks much much better
         print(Card(10,"hearts"))

10 of hearts

In [55]: cards = []
         for suit in [’Hearts’,’Spades’,’Diamonds’,’Clubs’]:
             for value in range(0,13):
                 cards.append(Card(value+1,suit))

In [56]: print(cards)
         print()
         str(cards[0])
         # now this make sense

[1 of Hearts, 2 of Hearts, 3 of Hearts, 4 of Hearts, 5 of Hearts, 6 of Hearts, 7 of Hearts, 8 of Hearts

Out[56]: ’1 of Hearts’

In [57]: from random import shuffle
         shuffle(cards)
         print(cards)

[3 of Diamonds, 4 of Hearts, 1 of Clubs, 4 of Clubs, 6 of Clubs, 7 of Spades, 5 of Clubs, 8 of Diamonds

In [58]: sorted(cards)

Out[58]: [1 of Clubs,
          1 of Diamonds,
1 of Hearts,
1 of Spades,
2 of Spades,
2 of Diamonds,
2 of Clubs,
2 of Hearts,
3 of Diamonds,
3 of Hearts,
3 of Spades,
3 of Clubs,
4 of Hearts,
4 of Clubs,
4 of Spades,
4 of Diamonds,
5 of Clubs,
5 of Hearts,
5 of Diamonds,
5 of Spades,
6 of Clubs,
6 of Hearts,
6 of Spades,
6 of Diamonds,
7 of Spades,
7 of Clubs,
7 of Hearts,
7 of Diamonds,
8 of Diamonds,
8 of Clubs,
8 of Spades,
8 of Hearts,
9 of Hearts,
9 of Diamonds,
9 of Spades,
9 of Clubs,
10 of Diamonds,
10 of Spades,
10 of Hearts,
10 of Clubs,
11 of Diamonds,
11 of Spades,
11 of Clubs,
11 of Hearts,
12 of Clubs,
12 of Hearts,
12 of Diamonds,
12 of Spades,
13 of Spades,
13 of Hearts,
13 of Diamonds,
13 of Clubs]
```