

Class notes: Monte Carlo methods
Week 2 supplement on Python classes
Jonathan Goodman
September 16, 2020

If you do Monte Carlo in Python, or anything else real in Python, you are likely to use libraries or software created by Python experts. This software is likely to use Python classes. Therefore, most people who do Monte Carlo need to know how classes work in Python. The Python documentation is (in September of 2020)

<https://docs.python.org/3/tutorial/classes.html>

This is good and clear. I will say a few of the things there, which will be enough for this week. I will not talk about inheritance, which you will remember from C++ or Java if you did classes in those languages. I will not talk about Python specific class features. Just the basics.

This supplement describes the class `datum`, which is defined and illustrated in the modules `DatumClassDef.py` and `DatumCaller.py` that are posted for Week 2. The main program is in the module `DatumCaller.py`. This module imports `DatumClassDef.py` and creates a *namespace* `dat` for it. The `import` command executes the module `DatumClassDef.py`, which creates a class `datum`. After the `import` command, the namespace `dat` will contain the name `datum`, which is the definition of the `datum` class.

A *class* is a software defined data type. If `datum` is a class, then you can create and use objects whose type is `datum`. An object whose type is `datum` is an *instance* of the class `datum`. Once you have defined the `datum` class, you can *instantiate* new *instances* of that class. The code in the class definition specifies what information a class object (instance) has and what a class object can do. The command `x = dat.datum(3, "x value")` creates a name `x` and binds this name to an object of class `datum` that is in the namespace `dat`. Please follow along in a line by line description of the code and the output. The class definition is figure 1. The code that illustrates how this works is in figure 2. The output is in figure 3.

Line 16 of figure 1 begins the definition of a class called `datum`. The next four lines are the `docstring`. All Python definitions (for this Monte Carlo class) must have docstrings. You can look up in Python style guides what a docstring is supposed to be like. The ones in this example are probably not “up to code”, but at least there’s something.

Line 22 defines a function called `__init__`. This function is the *constructor* for the class. It says how a new object from this class is to be created. The underscore characters before and after the name `init` is to prevent a programmer from accessing `init` from outside the class definition. The Python classes documentation linked above explains this in more detail. Python allows programmers to do things they should not do. Programmers have conventions to

keep themselves from doing things they should not do, such as putting underscore characters in names you're not supposed to use.

The first argument of `__init__` is a namespace called `self`. Each class instance has its own `self` namespace. This is where class instances put data that are unique to that class instance. The second argument is `val = 0`. This defines a variable `val` and gives it the default value 0. If you call the constructor without giving a value, the constructor uses the default value instead. Line 16 of `DatumCaller.py` in figure 2 creates a name `x` and binds it to an object of class `datum`. It creates this object by calling `__init__` with arguments 3 and "`x value`". The Python interpreter creates the `self` object and stores it with the class instance object. Lines 26 and 27 of the constructor (figure 1) has the class instance "remember" the given values.

Classes not only remember stuff, they do stuff. Line 29 of the class definition (figure 1) defines a function `get_value`. In the function definition it has the `self` argument. But it is called from line 18 in figure 2 with no arguments. The Python interpreter inserts the `self` object associated to that object. Line 19 of figure 2 illustrates that a function in a class can modify the data in the class. This one changes `self.val`, as instructed by line 39 of figure 1. Line 21 of figure 2 verifies that this happened. Line 22 of figure 2 illustrates that the a function that modifies class instance data can have arguments. There is one argument in line 22 but two arguments in line 41 of figure 1.

Lines 29 - 31 of figure 2 illustrate the feature that assignment in Python does not create objects, it only binds names to existing objects. Saying `y=x` doesn't create an object and copy the information from `x`. It just creates a name `y` and has `y` point to the same object the name `x` points to. Lines 26 and 27 illustrate that `y` has the same data as `x`, which you expect. Line 34 illustrates that changing the `y` value has the *side effect* of changing the `x` value. Line 33 accessed the object that name `y` points to. Line 34 calls `x.get_val()`. What really happens is that the names `y` and `x` are just different ways to get to the same object.

We in Monte Carlo create classes to store data and evaluate probability densities or do other big jobs. Line 41 of figure 1 creates a function that combines "user input" (the argument `x`) with information in the class itself to produce a value, which is the "capped" value `min(x, val)`.

Exercise

(not to hand in) Add a data member `self.sig` that stores a non-negative floating point number σ that represents the uncertainty in `val`. Create a test function to see whether a given number is within the error bar of the data value. More specifically:

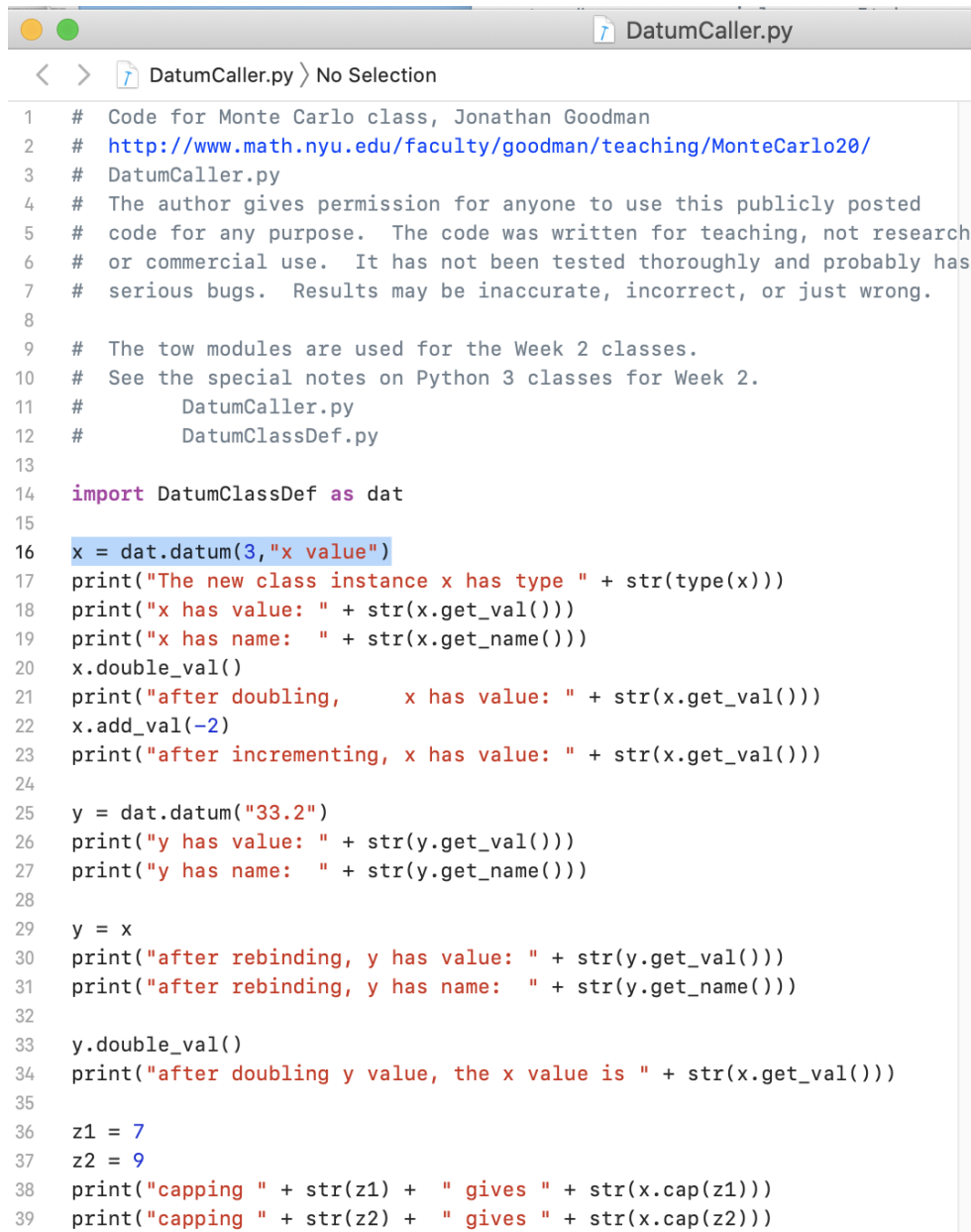
- Add an argument to the constructor `__init__` that copies the given argument to the `self` namespace. Give this default value $\sigma_{\text{default}} = 0$, though any scientist would tell you that's wrong because no data number has zero uncertainty.

- Add a `getter` function that returns σ stored in `self` (stored in the object bound to a name in `self`). The existing functions `get_val()` and `get_name()` are getter functions. Make one for `sig`.
- Add code to `DatumCaller.py` to test that these work.
- Create a class function `we(y, z)` (for “within error”) to test whether y is within z standard deviations (σ 's) of the data value. `x.we(y,z)` should return `TRUE` if $|y - \text{value}_x| \leq z\sigma_x$, and `FALSE` otherwise. Give z the default value $z_{\text{default}} = 1$, for a Monte Carlo one standard deviation error bar.
- Add code to `DatumCaller.py` that checks both the `TRUE` and the `FALSE` outcomes.

```
1 # Code for Monte Carlo class, Jonathan Goodman
2 # http://www.math.nyu.edu/faculty/goodman/teaching/MonteCarlo20/
3 # DatumClassDef.py
4 # The author gives permission for anyone to use this publicly posted
5 # code for any purpose. The code was written for teaching, not research
6 # or commercial use. It has not been tested thoroughly and probably has
7 # serious bugs. Results may be inaccurate, incorrect, or just wrong.
8
9 # The tow modules are used for the Week 2 classes.
10 # See the special notes on Python 3 classes for Week 2.
11 # DatumCaller.py
12 # DatumClassDef.py
13
14
15 class datum:
16     """A class to represent a floating point number associated to a name.
17     A 'datum' (from a Latin word for 'given') is one numerical piece of
18     information that one would use in statistics. More than one datum
19     is 'data'. One datum, two data.
20     """
21
22     def __init__(self, val = 0., name = "blank"):
23         """Create a datum with given value and name
24         record the given value and name"""
25
26         self.val = float(val) # Cast the given value to float
27         self.name = name
28
29     def get_val(self):
30         """return the value"""
31         return self.val # Don't need comments for obvious things
32
33     def get_name( self):
34         """return the name"""
35         return self.name
36
37     def double_val( self):
38         "double the value"
39         self.val = 2.*self.val
40
41     def cap( self, x):|
42         """cap x by not letting it be larger than the data value
43         Return the minimum of x and the value."""
44         if (self.val < x): # the data value if x is larger
45             return self.val
46         return x
47
48     def add_val( self, increment):
49         """add the increment to the value and store the new value"""
50         self.val = self.val + float(increment)
51
```

Figure 1: Code from DatumClassDef.py, showing line numbers.

fig:def



```
1 # Code for Monte Carlo class, Jonathan Goodman
2 # http://www.math.nyu.edu/faculty/goodman/teaching/MonteCarlo20/
3 # DatumCaller.py
4 # The author gives permission for anyone to use this publicly posted
5 # code for any purpose. The code was written for teaching, not research
6 # or commercial use. It has not been tested thoroughly and probably has
7 # serious bugs. Results may be inaccurate, incorrect, or just wrong.
8
9 # The tow modules are used for the Week 2 classes.
10 # See the special notes on Python 3 classes for Week 2.
11 # DatumCaller.py
12 # DatumClassDef.py
13
14 import DatumClassDef as dat
15
16 x = dat.datum(3, "x value")
17 print("The new class instance x has type " + str(type(x)))
18 print("x has value: " + str(x.get_val()))
19 print("x has name: " + str(x.get_name()))
20 x.double_val()
21 print("after doubling, x has value: " + str(x.get_val()))
22 x.add_val(-2)
23 print("after incrementing, x has value: " + str(x.get_val()))
24
25 y = dat.datum("33.2")
26 print("y has value: " + str(y.get_val()))
27 print("y has name: " + str(y.get_name()))
28
29 y = x
30 print("after rebinding, y has value: " + str(y.get_val()))
31 print("after rebinding, y has name: " + str(y.get_name()))
32
33 y.double_val()
34 print("after doubling y value, the x value is " + str(x.get_val()))
35
36 z1 = 7
37 z2 = 9
38 print("capping " + str(z1) + " gives " + str(x.cap(z1)))
39 print("capping " + str(z2) + " gives " + str(x.cap(z2)))
```

Figure 2: Code from DatumCaller.py, showing line numbers.

fig:call

```
[[JonathansMBP20:MonteCarlo20/classMaterials/Week2] jg% python3 DatumCaller.py
The new class instance x has type <class 'DatumClassDef.datum'>
x has value: 3.0
x has name: x value
after doubling, x has value: 6.0
after incrementing, x has value: 4.0
y has value: 33.2
y has name: blank
after rebinding, y has value: 4.0
after rebinding, y has name: x value
after doubling y value, the x value is 8.0
capping 7 gives 7
capping 9 gives 8.0
```

Figure 3: Output from running DatumCaller.py.

fig:output