Scientific Computing

Classes in Python 3

Jonathan Goodman, Fall, 2022

Introduction to Classes in Python 3

Numerical software often takes the form of a "package". You "tell your problem" to the package and the package applies the solution algorithm. The problem can be communicated to the package in the form of a class *instance*. That instance *object* contains the code and the data defining the problem. The package accesses this information by calling a class *attribute* that evaluates the function or functions defining the problem.

Programming languages define data *types*, such as *integer* or *string*. In Python, each *object* has a type, which can be *built in* (defined by the Python language), defined by a package (such as an `ndarray` in the *numpy* class), or defined by you, the programmer. A *class* is a defined datatype (or data type?). An *instance* of that class is an object whose datatype is that class. Creating a new object from a class is *instantiating* a class object. It can be complicated, for sophisticated classes, but it can be simple too.

The best introduction to Python classes (my opinion, others disagree!) is in the Python 3 documentation itself. This is not a "for dummies" collection of recipes, but an explanation of principles. It explains not only classes, but other basic principles of Python. Click on the link and see.

Python 3 classes documentation

An example

The sample code `ClassDemo.py` illustrates the class mechanism. It shows how simple classes are in Python. To run it, you also need the files (modules) `BlankSlate.py`, `GraySlate.py`, and `Helpers.py`. Copy all of these into a directory and then type `python ClassDemo.py`. The output should be identical to the screenshot `ClassDemoOutput.png`. Open the Python modules in editor windows that show line numbers. Lines 9 to 11 of `ClassDemo.py` illustrate that a class is defined in a module, such as `BlankSlate.py` that you import if you want to instantiate (create) an object of that class. Line 17 creates an object, `BlankInstance` (more correctly, a *name*, "`BlankInstance`", that is bound to an object of that class). The `bs.`$\cdots$ on right side of `=` says that you are looking for a name in the *namespace* "`bs`". Line 9 of `BlankSlate.py` says that "`BlankSlate`" is a class. Line 17 of `ClassDemo.py` has $\cdots$`BlankSlate()`, which uses the *constructor* of the `BlankSlate` class to create (instantiate) an object of that class.

Every class has a constructor, which is a function with the same name as the class, BlankSlate in this case. But the class definition in BlankSlate.py does not include any constructor code, so Python uses the *default constructor*. The constructor, possibly among other things, creates a namespace for the class instance.

As the documentation explains, a *namespace* is a *dictionary* of names and objects. The command dir returns a list of all the names of this dictionary. You can see in the output that there are quite a few, and that all of them have the format __SomeName__. The underscore characters "__" give the message: "you are allowed to access these, but please don't unless you really know what you're doing." For example, if your code has BlankInstance.__hash__, it is a mistake in your judgement, but not a Python error. The module Helptes.py returns only names without the underscore warning. Lines 23 to 26 show that there aren't any. The BlankSlate class has nothing in it.

Line 28 creates a new name in the BlankInstance namespace, which is a function in the h namespace defined by importing the Helpers module. Lines 30 to 34 illustrate that this name was added to the namespace of BlankInstance. This name was not added to the BlankSlate class, only to the instance of that class BlankInstance. Line 37 shows that you can access that name. A numerical software package accesses its "problem" function in this way. In C++ or Java, you cannot insert a new member to a class instance as line 28 does. The simplicity of Python, which is a major strength of the language, allows this. A class instance is not much more than a namespace dictionary.

The GraySlate.py module defines a GraySlate class that has an __init__() function. This function is called whenever a new class instance is created. Line 43 of ClassDemo.py creates a GraySlate instance with input=3. Note that the name "GraySlate" (the name of the class) is in the gs namespace that is created by line 10. Using namespaces allows the same name to have different meanings in different places. In line 10, GraySlate refers to the module (file) GraySlate.py. A class instance has a personal namespace traditionally called self that is created by the constructor. Line 15 of GraySlate.py "remembers" the input n by copying its value into the instance namespace. The self namespace holds data associated to a class instance. Lines 17 1nd 18 define a *class function* add_n that is a member of any class instance. Lines 48 to 50 of ClassDemo.py show that the names were added to the instance GS3 when that object was built. Lines 52 to 54 show that the class instance GS3 remembered the input value 3 that was supplied in line 43. The name n in the instance namespace (accessed by GS3.n in line 53) is bound to that value.

The code in the class definition module GraySlate.py (the code defining the GraySlate class starts at line 10) needs a way to access the namespace associated to a class instance. That namespace is the first argument to any function defined within the class (i.e., after line 10). A good Python programmer will call that self, as the __init__() function does. A bad programmer can call it whatever he/she/they wants, as line 17 shows.

Lines 17 and 18 of GraySlate.py show how data "known" by a class instance myself.n can be combined with input data x to define a function. A function that defines a numerical problem to a numerical software package can do it this way. This makes Python classes a convenient way to define computational problems to packages.

Lines 61 to 63 of `ClassDemo.py` illustrate the difference between *names* and *objects* in Python. Line 61 creates a new name, `newGI` (for "new Gray Instance") but does not create a new object. Instead, the new name is bound to the same object that `GI3` was bound to. The names `newGI.n` and `GI3.n` point to the same object and therefore have the same value.

A package and a function for it

The module `IntegrationDemo.py` shows how a class can hold data that defines a function. This demo concerns

$$\int_{-1}^{1} \left(1 + 2x + 3x^2\right) dx = 4 \; .$$

The integrand $f(x) = 1 + 2x + 3x^2$ is represent by an instance of the `Integrand` class defined in the module `IntegrandClass.py`. The integration "package" is in the module `IntegrationPackage.py`. Line 19 of `IntegrationPackage.py` has the integration package accessing the function, as the `f` attribute of the `f` object. You can recognize lines 18 and 19 as the rectangle rule, which is not very accurate (to see this, type: `python IntegrationDemo.py`).

Lines 13 to 17 of `IntegrationDemo.py` create the information that defines the integrand, which is a polynomial of degree 2 with coefficients $c + 0 = 1$, $c_1 = 2$ and $c_2 = 3$. Line 19 creates a class instance object which holds this information. Lines 21 to 29 show that the `f` attribute of the `poly` object, which is accessed in `poly.f(x)` in lines 23 and 28, evaluates the polynomial correctly, at least for $x = 0$ and $x = 2$. Line 38 demonstrates calling the integration package. It needs an object, `poly` in this case, that has a `f` attribute. This object contains the data and the code that defines $f(x)$.

The integrand class module `IntegrandClass.py` has an `__init()__` that is slightly fancier than `GraySlate.py`. This one does error checking. It also makes a local copy of the data, done in line 21. That way, the integrand doesn't change if some code elsewhere changes the array `coeffs`. Lines 30 and 31 illustrate the instance object accessing data that the integration package doesn't know or want to know.