Scientific Computing, Courant Institute, Fall 2021
http://www.math.nyu.edu/faculty/goodman/teaching/SciComp2021/index.html

Python Part 2

Jonathan Goodman, September, 2021

1 About these notes

These notes explain parts of Python that allow you to define a function that depends on a dataset and use it in a scientific computation. The person designing scientific computing software will not know precisely what data is needed to define a specific function, so he/she wants software that works with a function without seeing the data. Many software packages work in this way, including the **Emcee**, a code for Bayesian estimation and uncertainty quantification.

The Python *class* mechanism allows for such data storage and hiding. A *class* in Python is a user-defined datatype. You define a class designed to hold data, then you create an *instance* of the class, which is a Python object, for the specific dataset. The *constructor* of the class (see below) reads the data, and makes it available to other functions in the class, when associated with this specific instance. Other functions associated to the class say how the function is defined in terms of the data.

These notes explain a minimum you need to do this. There is some general background about classes, but it is better to get that straight from the **Python documentation**. Of course `stack exchange` has reliable answers to many specific questions. As usual, I discourage people from using "for dummies" explanations, which generally tell you how to do very specific things without telling you how to modify them to do what you really want.

Many aspects and uses of classes are not mentioned here. One big topic not mentioned is the distinction between *instance variables* and *class variables*. Here, class variables (or names that are bound to them) are not discussed at all. There's nothing here about class *inheritance* either.

I appreciate that the material seems unnecessarily technical. But it's not really that technical because it's based on just a few powerful principles. It is possible for ordinary Scientific Computing students to understand enough to create and use classes like a pro. You don't have to work by trial and error from examples. The Python *class* mechanism is *lightweight*. This means it is not complicated and doesn't take a lot of code to accomplish. It helps that it is based on simple general principles rather than a long list of rules for particular cases.

2 Classes = functions acting on data

Modular code is a core software principle. Code for separate tasks should be isolated into separate modules where it can be developed and tested without the clutter of a complex and changing context. Interfaces between different modules or software components should be designed so that each part "sees" what it needs to do its function.

This principle suggests that part A of a code should be able to use part B without telling part B data that part A doesn't use itself. For example, suppose part A computes the integral of a function $f(x)$ whose values are computed by part B. The function f probably depends on more data than just the argument x. It could be as simple as $f(x) = x^n$, with a single data number n in addition to the argument x.

The code that evaluates the integral needs to communicate x values to the code that evaluates f, but it should not communicate, or know, the value of n. That's *data hiding*. This simplifies the integration code. It also makes it possible to test the integration code on simple functions that don't need much data and then apply it to complex functions that require large datasets. Modular validation (debugging) is as important as modular code.

Classes are a way to create modular code that does data hiding and allows modular validation. By definition, a class is a data type defined by software. The definition can be in a package, such as `numpy`, or you write your own code to define your own class. If you define a class called `A` then the environment can have objects whose type is `A`. An object `x` of type `A` in an *instance* of the class `A`. Names may be bound to class instances (objects of type `A`) in the usual way. As with other objects, the command $y = x$ does not copy the contents of the object `x` points to. It just binds the name `y` to the same object.
Warning. The class mechanism is Python 3 is different from what it was in Python 2.7 and is radically different from the mechanism in C++ or Java. One difference between C++ and Python classes depends on the fact that Python classes involve namespaces that anyone can add to. Python classes have no privacy. You can add a name to the namespace of an instance of a class without adding that name to the class definition. The Python programmer must take care not to abuse this freedom in a way that makes unreadable code.

3 Scope

A *scope* is a block of Python code where a local namespace is defined. You can read about this in the Python documentation link above. The code inside a function or class definition is a scope. Any name defined inside a scope and added to the local namespace will be lost when the code leaves that scope. Objects these names are bound to might be forgotten if no other names are bound to them. The code defining a function or a class is a scope.

4 Functions

A *function* in Python is a kind of object. It consists of an argument list and a sequence of Python commands, which are the *body* of the function. The *keyword*[1] `def` tells the Python interpreter that a function is being defined, first giving a name and arguments, then the body. You access a function through a name bound to it, just as you access numbers or other objects. As with other objects, you can bind more than one name to the same function, though you should have a good reason to do that. Lines 28 and 29 in Figure 1 illustrate this. A function is evaluated by executing the commands in its body.

The arguments are passed from the calling code to the function in a "pythonic" way (a way that is natural in the Python mindset). The names in the argument list are added to the local namespace in the scope of the function definition. These names are bound to objects that are present or created when the function is called. New objects are created for immutable objects but not mutable ones.

Figure 1 shows the basics of Python functions, from a numerical computing point of view. Lines 10 to 19 are a scope that contains the commands that define the function. Line 9 creates the name `q` and binds it to this function. The `(x)` is the *argument list* (consisting of just one argument here) that contains the name `x`. Line 22 (among other things) binds the name `x` (from the argument list) to the object that name `t` is bound to when the command is executed. In this case, it's the immutable object `2.0`. Executing line 22 will cause lines 10 to 19 to be executed, with name `x` bound to object `2.0`. Line 10 is a *docstring.* In this class, every function and class definition must have a complete docstring. Lines 10 to 15 of Figure 2 give an example of a more complete docstring. In a perfect world, you would be able to use the function without reading the code in the function, only the docstring.

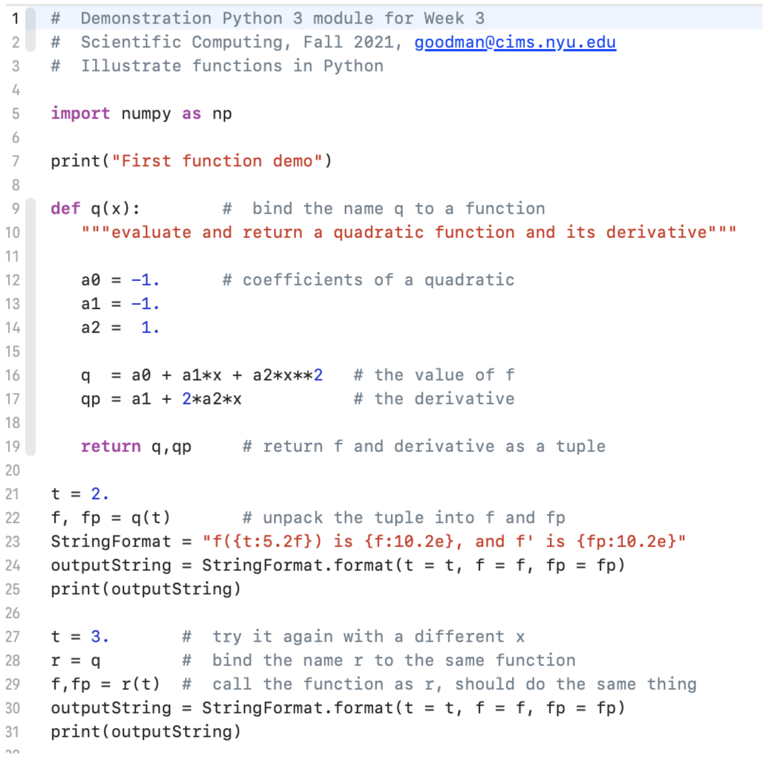
The `return` command in line 19 passes information back from the function. The names `q` and `qp` are bound to objects. The command binds names
f and `fp` (line 29) to these objects. The pair `q,qp` on line 19 form a *tuple.* A tuple is a Python object that acts like an un-named list (a list with no name bound to it). A tuple can have any number of names. The number of objects being broadcast (two, in line 19) must match the number of names being bound (two, in line 22). The output from lines 23 to 25 (first output line below) lets us check that the function $q(x) = x^2 - x - 1$ (the "Fibonacci polynomial" from Week 1) correctly evaluated to $q(2) =$ and $q'(2) = 3$.

Lines 28 and 29 illustrate the fact that `q` is a name bound to the function defined in lines 9 to 19. The name `r` can be bound to the same function. Normally, you would need a good reason to do this because it makes the code harder to read. The second line of output shows that, again correctly, $x^2 - x - 1$ evaluates to 5 when $x = 3$.

[1] A *keyword* is a name (character string) that has a pre-defined meaning to the interpreter. Other keywords are `if`, `return`, etc. You are not allowed to bind a keyword (assign an object to it). You will see this if you try, as a command, `for = 4`, because `for` is a keyword.

```
jg% python3 FunctionDemo.py
First function demo
f( 2.00) is   1.00e+00, and f' is   3.00e+00
f( 3.00) is   5.00e+00, and f' is   5.00e+00
```

A function must be defined before it can be used. That's why the function]tt q in Figure 1 is defined at the beginning and used at the end. It is usually recommended to define functions in other modules instead, and read in their definitions using `import` statements. That makes the code in the "main program" easier to figure out, because the main things (starting at line 21) would be at the top rather than after some technical function definitions.



```
1   #  Demonstration Python 3 module for Week 3
2   #  Scientific Computing, Fall 2021, goodman@cims.nyu.edu
3   #  Illustrate functions in Python
4
5   import numpy as np
6
7   print("First function demo")
8
9   def q(x):        #  bind the name q to a function
10     """evaluate and return a quadratic function and its derivative"""
11
12     a0 = -1.      # coefficients of a quadratic
13     a1 = -1.
14     a2 =  1.
15
16     q  = a0 + a1*x + a2*x**2   # the value of f
17     qp = a1 + 2*a2*x           # the derivative
18
19     return q,qp     # return f and derivative as a tuple
20
21  t = 2.
22  f, fp = q(t)       # unpack the tuple into f and fp
23  StringFormat = "f({t:5.2f}) is {f:10.2e}, and f' is {fp:10.2e}"
24  outputString = StringFormat.format(t = t, f = f, fp = fp)
25  print(outputString)
26
27  t = 3.       #  try it again with a different x
28  r = q        #  bind the name r to the same function
29  f,fp = r(t)  #  call the function as r, should do the same thing
30  outputString = StringFormat.format(t = t, f = f, fp = fp)
31  print(outputString)
```

Figure 1: Code illustrating the basic mechanics of Python functions.

Figure 2 illustrates a few more things. This function has two arguments, one being a number and the other a numpy array. The docstring documents the function, so you can use the function after reading only the docstring. Unfortunately, real docstrings don't live up to this standard. For example, this one doesn't tell the user how the function knows the degree of the polynomial. You have to read the code to see that it gets it from the number of coefficients, which is $\deg(p) + 1$ (see line 17). A careful grader would take off a point for a mistake like that. In a collaborative coding environment, your co-worker would complain to you.

Lines 39 to 42 define the specific polynomial:

$$p(x) = x^d + 2x^{d-1} + \cdots + dx + (d+1) \ .$$

Here $d = 4$.

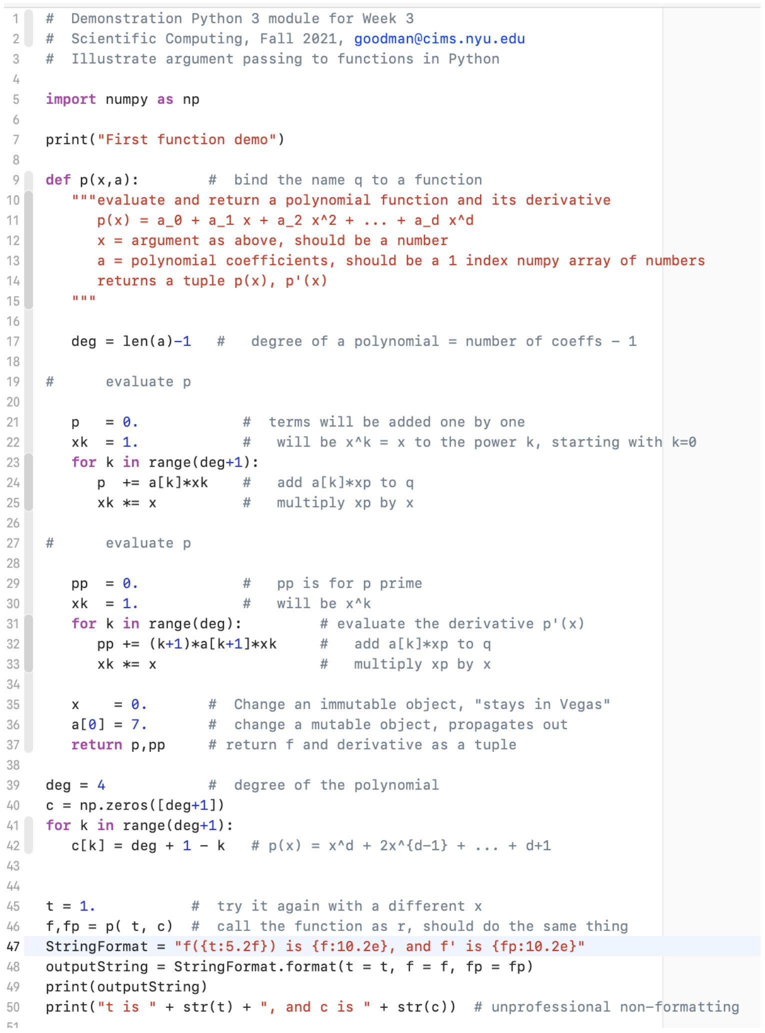
When `p` is called from line 46, the name `a` is bound to the object that the argument `c` was bound to. In this case, that's a numpy array. This is an example of what in C/C++/FORTRAN is called *call by reference*. If such a mutable object is "mutated" (changed) in the function, that can be seen from outside the function. It would be a *side effect* of the function. Communicating from the function using side effects can make code hard to follow, particularly if these side effects are not described clearly in the docstring. If you re-assign an immutable object as in line 35, that effect is not seen outside the function[2] because the new object (`7.0` in this case) is only bound to from a name in the namespace of the function. Passing immutable objects to functions acts like *call-by-value* in C/C++/FORTRAN.

Here is the output, from the command line:

```
jg% python3 ArgumentPassing.py
First function demo
f( 1.00) is   1.50e+01, and f' is   2.00e+01
t is 1.0, and c is [7. 4. 3. 2. 1.]
```

First, you see that it correctly calculates $p(1) = 1 + 2 + \cdots + 5 = 15$ and $p'(1) = 4 + 6 + 6 + 4 = 20$. Next you see that the immutable argument, which was re-assigned in the function (line 35) kept its value outside. You see that line 36 did alter the array `c`. The value $c[5]$, which had been 5 was changed to 7. The unformatted output here is not so hard to read, which is only because the entries of c are simple numbers. The grader, if she/he were were grading very carefully, would subtract a point for using unformatted output. I did it to show that you do it in debugging and experiments, but not in the final finished product.

[2]There is a saying "What happens in Vegas stays in Vegas." This refers to something someone might do in the gambling and prostitution capital of the US, Las Vegas. You're not supposed to tell anyone about it when you get back.



```python
#  Demonstration Python 3 module for Week 3
#  Scientific Computing, Fall 2021, goodman@cims.nyu.edu
#  Illustrate argument passing to functions in Python

import numpy as np

print("First function demo")

def p(x,a):        #  bind the name q to a function
   """evaluate and return a polynomial function and its derivative
      p(x) = a_0 + a_1 x + a_2 x^2 + ... + a_d x^d
      x = argument as above, should be a number
      a = polynomial coefficients, should be a 1 index numpy array of numbers
      returns a tuple p(x), p'(x)
   """

   deg = len(a)-1   #   degree of a polynomial = number of coeffs - 1

#      evaluate p

   p   = 0.            #  terms will be added one by one
   xk  = 1.            #   will be x^k = x to the power k, starting with k=0
   for k in range(deg+1):
      p  += a[k]*xk    #   add a[k]*xp to q
      xk *= x          #   multiply xp by x

#      evaluate p

   pp  = 0.            #   pp is for p prime
   xk  = 1.            #   will be x^k
   for k in range(deg):         # evaluate the derivative p'(x)
      pp += (k+1)*a[k+1]*xk     #   add a[k]*xp to q
      xk *= x                   #   multiply xp by x

   x    = 0.       #  Change an immutable object, "stays in Vegas"
   a[0] = 7.       #  change a mutable object, propagates out
   return p,pp     # return f and derivative as a tuple

deg = 4            #  degree of the polynomial
c = np.zeros([deg+1])
for k in range(deg+1):
   c[k] = deg + 1 - k   # p(x) = x^d + 2x^{d-1} + ... + d+1


t = 1.           #  try it again with a different x
f,fp = p( t, c)  #  call the function as r, should do the same thing
StringFormat = "f({t:5.2f}) is {f:10.2e}, and f' is {fp:10.2e}"
outputString = StringFormat.format(t = t, f = f, fp = fp)
print(outputString)
print("t is " + str(t) + ", and c is " + str(c))  # unprofessional non-formatting
```

Figure 2: Code illustrating function argument passing.

5 Classes

The material in this section may seem technical. I don't know it's described in the clearest way. But once you get it, you will be able to use classes confidently. They are simple, but

A *class* is a user defined datatype. Defining a class (details below) has the effect of creating a name that is bound to the definition of the class. If, for example, the name `A` is bound to a class, we call the class `A`. The Python command

```
x = A(args)
```

creates a new object of type `A` using information in the argument list `args`, and binds the name `x` to this class instance object. Some of the code defining the class (the function `__init__` described below) says how the new object is to be built.

A class object (instance of the class) may hold data that is unique to that instance. It accesses this data using names in a dedicated namespace for the instance. The interpreter keeps track of all these namespaces for all the objects of a given class and supplies the right one to each instance as needed. The namespace has two names, depending on whether you are "inside the instance" or not. Outside the class, the namespace is the name bound to the class object. For example, the name `x.f` refers to the name `f` in the instance `x`. The code defining the class cannot work this way because the name of the object (class instance) is not known when the class is defined and different instances will have different names. By tradition, "`this`" is the name of the namespace in code defining the class.

Figure 3 illustrates definition of a class. The class is used for data fitting of an oscillation to data The data are n observations X_i made at times t_i. The theory/model is a simple oscillation $f(t) = A\sin(\omega(t - t_0))$. The mismatch is

$$R^2(\omega, A, t) = \sum_{i=1}^{n} \left[X_i - A\sin(\omega(t_i - t))\right]^2 \ . \qquad (1)$$

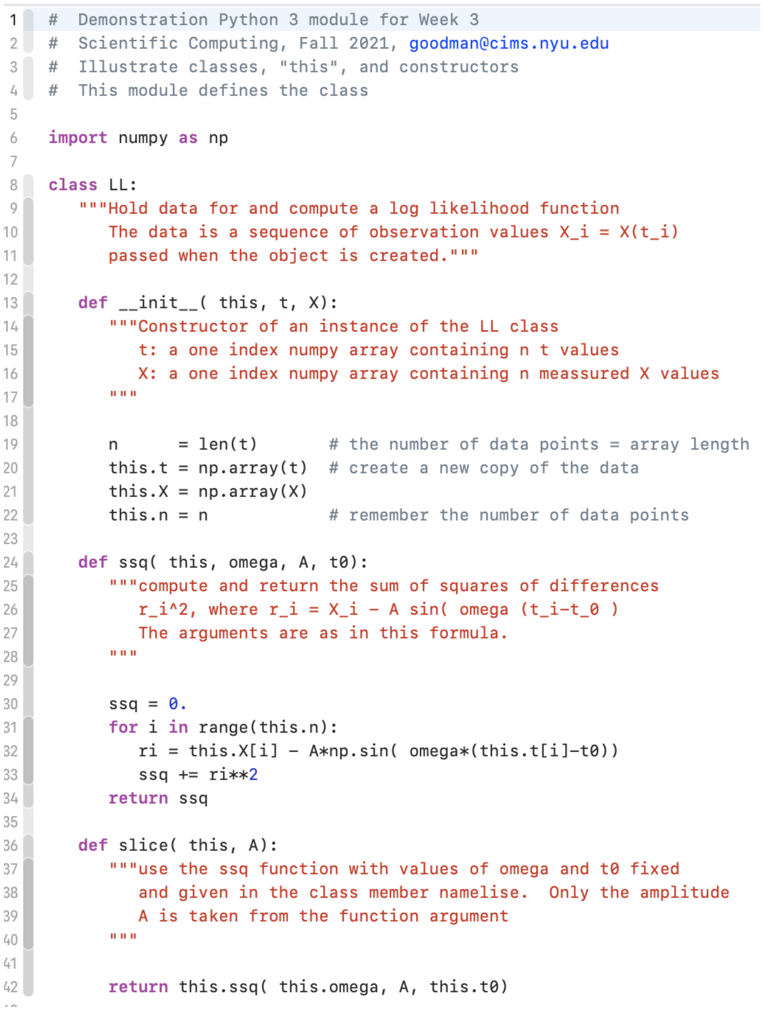
This sum of squares is the log of the likelihood function, assuming a Gaussian error model and neglecting irrelevant constants.

Line 8 says we're defining a class whose name is `LL` (for Log Likelihood). Then comes three lines of docstring. Please decide for yourself whether the docstring meets the standard that you can use the class after reading the docstring but without reading the rest of the code defining the class. Line 13 defines a function called `__init__`. The underscore characters before and after `init` are a Python convention that tells you not to call this function yourself. Only the developer (or, in this case, the Python interpreter) should call it. The `__init__` function is the *constructor* for class instances. It says how an instance of thee `LL` class is to be built. The interpreter calls `__init__` for the class whenever a new instance of that class is created.

The three arguments to the constructor, `__init__`, go into the local namespace when `__init__` is executed. The first argument, `this`, is the name of the namespace of class instances (see below). It is created by the "system" (Python interpreter). The other two are arguments given to the constructor when it is called. Lines 22 and 31 of Figure 4 have calls to the constructor. Executing them creates two instances of the `LL` class. The names `model1` and `model2` are bound to these two instances. You can think of the function `LL` in the namespace `LL` as being bound to the constructor function `__init__`, but the reality probably is more complicated. The calls to the constructor in Figure 4 have just two arguments, `times` and `values`. The "third" argument (the first, actually), is `this`, which is supplied by the system. Inside the `__init__` function (lines 13 to 22 in Figure 3) the local namespace is "born" containing `this`, `t`, and `X`. Line 20 creates a copy of the input array `t` that is bound to from a name in the `this` namespace. If the command had been: `this.t = t`, it would have bound the local name to the same object the argument is bound to. That would make it possible for the calling routine in Figure 4 to change the data in the `model1` instance. To protect the local data, line 20 creates a new numpy array with the same data. Line 22 copies n from the local namespace to the `this` namespace. Names in the local namespace are forgotten when the `__init__` routine finishes, but names in `this` remain a part of the object being created. Line 31 shows that this name is still bound to n. The simpler expression: `for i in range(n):` would have been a bug because `n` is not a name in the local namespace of the function `ssq`.

Lines 24 to 34 of Figure 3 create a function that is part of the `LL` class. It computes the sum of squares using data X_i and t_i attached to this instance, using the supplied arguments ω, A, and t. It also has access to the namespace `this`. The data are stored in numpy arrays that are bound from names in `this`. Line 32 uses these values.

Finally, the *slice* function produces a one variable "slice" of the log likelihood function with all arguments except the amplitude, A fixed. This will allow the integration function to integrate over the one dimensional slice. The values of ω and t_0 are supposed to be known. You might wonder *how* they come to be known, given that the constructor doesn't set them. That is explained below. Line 62 evaluates the `ssq` function associated to the same class instance that called `slice`. The namespace `this` has a reference to the instance of the function with access to the correct data.



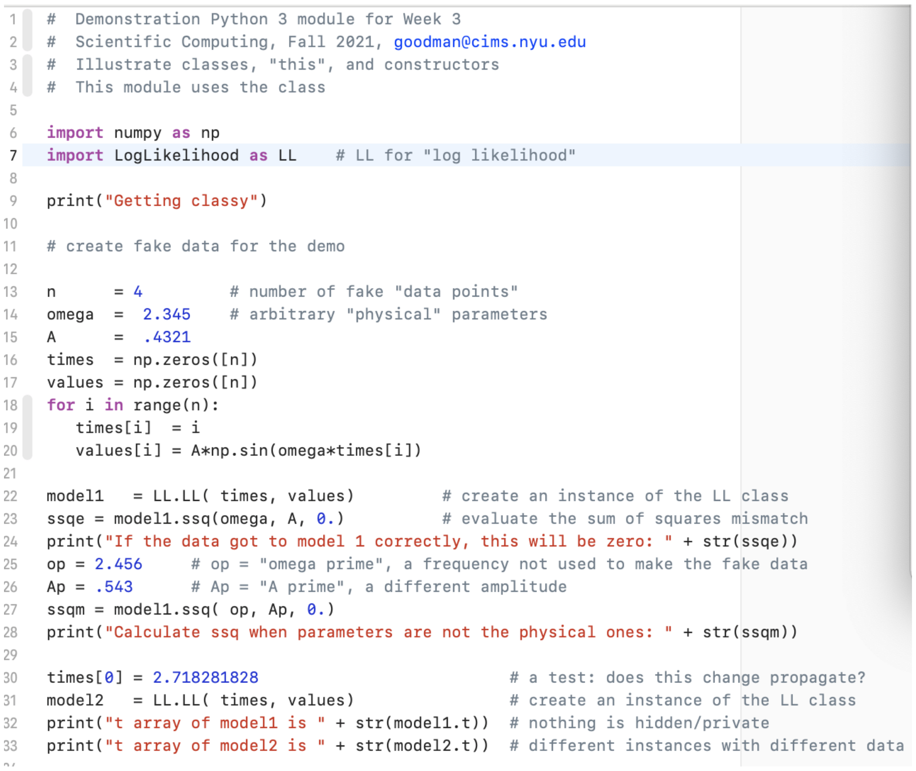
```
#  Demonstration Python 3 module for Week 3
#  Scientific Computing, Fall 2021, goodman@cims.nyu.edu
#  Illustrate classes, "this", and constructors
#  This module defines the class

import numpy as np

class LL:
   """Hold data for and compute a log likelihood function
      The data is a sequence of observation values X_i = X(t_i)
      passed when the object is created."""

   def __init__( this, t, X):
      """Constructor of an instance of the LL class
         t: a one index numpy array containing n t values
         X: a one index numpy array containing n meassured X values
      """

      n      = len(t)       # the number of data points = array length
      this.t = np.array(t)  # create a new copy of the data
      this.X = np.array(X)
      this.n = n            # remember the number of data points

   def ssq( this, omega, A, t0):
      """compute and return the sum of squares of differences
         r_i^2, where r_i = X_i - A sin( omega (t_i-t_0 )
         The arguments are as in this formula.
      """

      ssq = 0.
      for i in range(this.n):
         ri = this.X[i] - A*np.sin( omega*(this.t[i]-t0))
         ssq += ri**2
      return ssq

   def slice( this, A):
      """use the ssq function with values of omega and t0 fixed
         and given in the class member namelise.  Only the amplitude
         A is taken from the function argument
      """

      return this.ssq( this.omega, A, this.t0)
```

Figure 3: Code that defines a Log Likelihood class.

Figure 4 shows how the `LL` class is used. Line 7 imports the names defined in the module LogLikelihood into a namespace called `LL`. Lines 12 to 20 create numpy arrays `times` and `values` stuffed with fake data. Line 22 creates the name `model1` and binds it to an object of type `LL`. The data in numpy arrays `times` and `values` will be stored locally in the object, so that later changes to `times` do not propagate to (change) the object.

Lines 23 and 27 show the point of putting data into class objects in this way. Suppose we are trying to find parameters ω, A, and t to fit the data. This requires us to “query” the sum of squares function (1) many times with different parameter values. This might mean using a numerical integration or optimization package. We want to write integrators and optimizers that don’t know what kind of data their functions work on. Line 22 passes the data to the `model1` object. After that, the sum of squares may be evaluated for different parameter combinations without referring to the data.

The rest of the code in Figure 4 makes other pedagogical points. Line 31 shows that you can create more than one object of the same type. That could be a way to compare predictions from different data sets. Lines 32 and 33 illustrate that the datasets in `model1` and `model2` are different. They also illustrate the fact that data members in a class are not *private* and they can be in C++. [If you don't know C++, ignore this point.] Even though the developer of the Figure 4 code has direct access to the model data, he/she normally would not access it that way. It's better to use the interface created by the developer who "build" the `LL` class. In Python, there are many things that you *can* do that it's best *not* to do except in very carefully thought out and documented ways.



```
 1  #  Demonstration Python 3 module for Week 3
 2  #  Scientific Computing, Fall 2021, goodman@cims.nyu.edu
 3  #  Illustrate classes, "this", and constructors
 4  #  This module uses the class
 5
 6  import numpy as np
 7  import LogLikelihood as LL    # LL for "log likelihood"
 8
 9  print("Getting classy")
10
11  # create fake data for the demo
12
13  n      = 4         # number of fake "data points"
14  omega  =  2.345    # arbitrary "physical" parameters
15  A      =  .4321
16  times  = np.zeros([n])
17  values = np.zeros([n])
18  for i in range(n):
19     times[i]  = i
20     values[i] = A*np.sin(omega*times[i])
21
22  model1   = LL.LL( times, values)         # create an instance of the LL class
23  ssqe = model1.ssq(omega, A, 0.)          # evaluate the sum of squares mismatch
24  print("If the data got to model 1 correctly, this will be zero: " + str(ssqe))
25  op = 2.456      # op = "omega prime", a frequency not used to make the fake data
26  Ap = .543       # Ap = "A prime", a different amplitude
27  ssqm = model1.ssq( op, Ap, 0.)
28  print("Calculate ssq when parameters are not the physical ones: " + str(ssqm))
29
30  times[0] = 2.718281828                          # a test: does this change propagate?
31  model2   = LL.LL( times, values)                # create an instance of the LL class
32  print("t array of model1 is " + str(model1.t))  # nothing is hidden/private
33  print("t array of model2 is " + str(model2.t))  # different instances with different data
```

Figure 4: Code showing how the LL class is instantiated and used.

Here is what happens when you run the demo at the command line.

```
jg% python3 ClassDemo.py
Getting classy
If the data got to model 1 correctly, this will be zero: 0.0
```

```
Calculate ssq when parameters are not the physical ones: 0.04549229947150629
t array of model1 is [0. 1. 2. 3.]
t array of model2 is [2.71828183 1.         2.         3.        ]
```

After `Getting classy`, the first two output lines show that the sum of squares is zero (as it should be) when the exact parameters are used but non-zero otherwise. Next we see some of the stored data from `model1` and `model2`. You can see that un-formatted numerical output can be hard to read. That's why this class usually requires numerical output in programming exercises to be formatted and aligned.

6 Using a class function

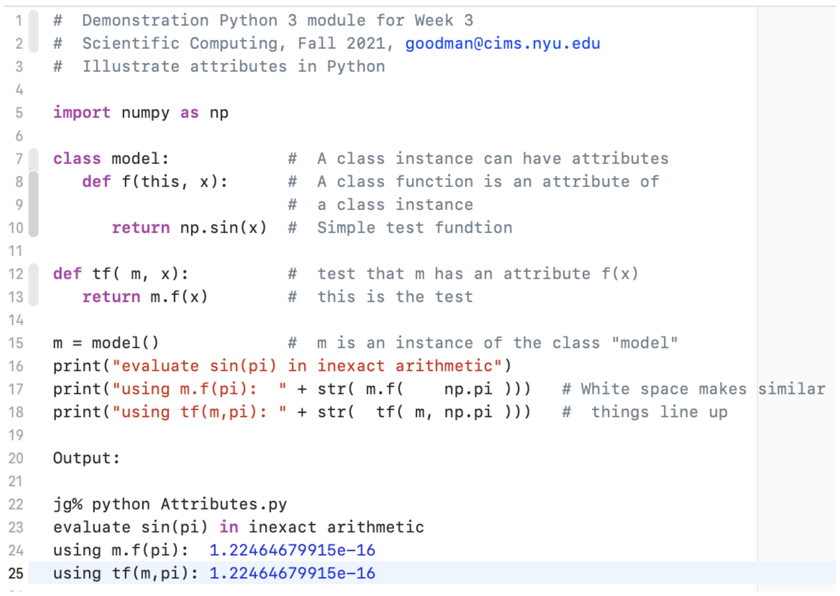
The Python concept of *attribute* of an object is similar to (the same as?) the concept of name in a namespace. If `m` is the name of an object, such as an instance of a class, then `f` is an attribute of `m` if the interpreter can find an object corresponding to `m.f`. This is the same (almost the same?) as `f` being in the namespace `m`, except that `m` can be an object that is not a namespace.

Figure 5 gives a basic illustration. The class `model` (line 7) has no constructor (no `__init__` function), just a function `f`. Line 8 shows that within the class, the function `f` has two arguments, with the first one being `this` (the instance namespace) and the second being the user supplied argument `x`. Lines 13 and 17 show that you just give the argument `x` when you use `f` from outside the class. Line 17 shows `f` being an attribute of `m`. The argument, a floating point approximation of π supplied by `numpy`, is passed to `f` in the usual way. Line 12 shows a function `tf` (for "text function") being defined with an argument `m`. This function knows nothing about `m` except that it has an attribute `f` (line 13). Line 18 shows passing an argument `m` to `tf` that does have an attribute `f`. The output shows that the second argument (π) is passed to `tf` and then to `m.f` to get the right result.

This attributes mechanism makes it possible to write flexible numerical code that takes functions defined in different ways. The function can be a simple formula, as here, or it can be defined in a complicated way using data, as in Figure 7. That object that "carries" `f` as an attribute might be simple, or it might have many other attributes and internal data. Python doesn't check the types of objects as they are passed to functions, as compiled languages like C++ or FORTRAN usually do. The definition of the function `tf` doesn't say anything about the types of the arguments `m` and `x`. The program will crash if `tf` tries to use one of its arguments in a way that is not defined or allowed. For example, it would crash if the argument `m` does not have an attribute `f`, or if `x` is a character string for which $\sin(x)$ is not defined.

I use test modules like the one in Figure 5 to debug my understanding of Python. I write code that is as simple as possible to explore and test whatever feature I'm interested in. It takes less time and is more clear when there's no other stuff "in the say", such as numerical integration software.

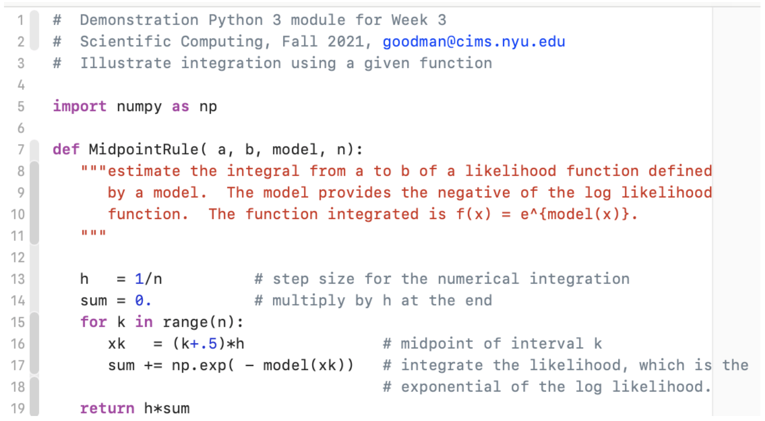
We will see (lines 35 and 36 of Figure 7) that the class definition does not necessarily determine what attributes a class instance might have. They can be added from the outside, which may be bad software practice because looking at the class definition doesn't tell you what's in the class.



```
#  Demonstration Python 3 module for Week 3
#  Scientific Computing, Fall 2021, goodman@cims.nyu.edu
#  Illustrate attributes in Python

import numpy as np

class model:               #  A class instance can have attributes
   def f(this, x):         #  A class function is an attribute of
                           #  a class instance
      return np.sin(x)     #  Simple test fundtion

def tf( m, x):             #  test that m has an attribute f(x)
   return m.f(x)           #  this is the test

m = model()                #  m is an instance of the class "model"
print("evaluate sin(pi) in inexact arithmetic")
print("using m.f(pi):  " + str( m.f(    np.pi )))   # White space makes similar
print("using tf(m,pi): " + str(  tf( m, np.pi )))   #  things line up

Output:

jg% python Attributes.py
evaluate sin(pi) in inexact arithmetic
using m.f(pi):  1.22464679915e-16
using tf(m,pi): 1.22464679915e-16
```

Figure 5: Use a class object to define a function to be integrated.

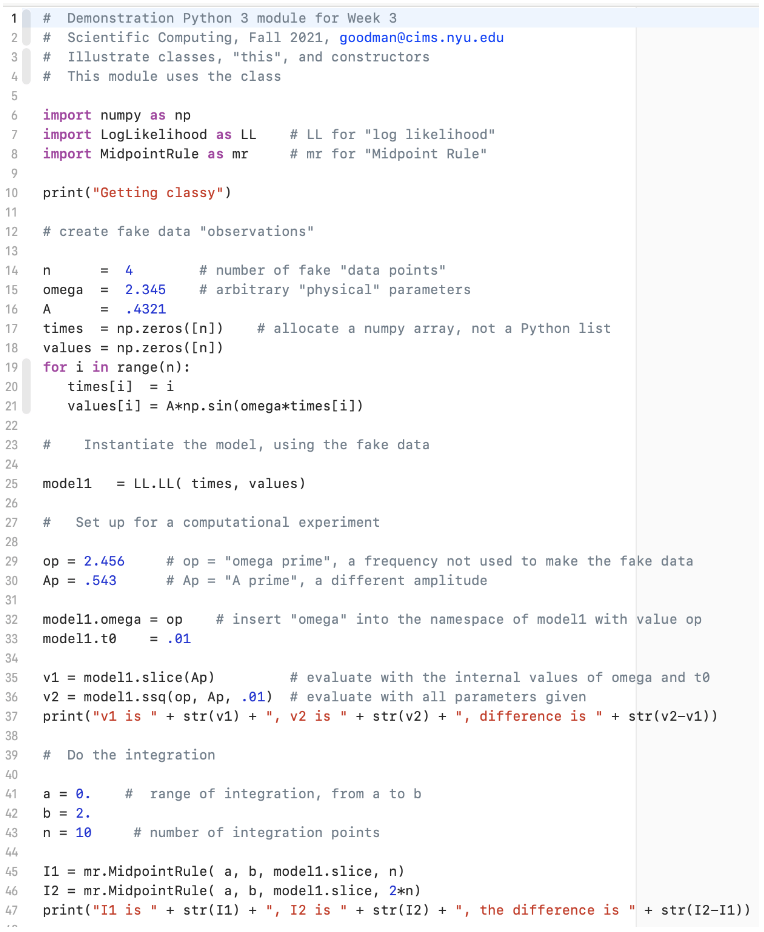
This section demonstrates code for numerical algorithm, with the code written for a generic function that is supplied as an argument. Figure 6 code implements the midpoint rule for integration. The docstring should explain this. The argument `model` is a function. Line 17 uses this function.



```python
#  Demonstration Python 3 module for Week 3
#  Scientific Computing, Fall 2021, goodman@cims.nyu.edu
#  Illustrate integration using a given function

import numpy as np

def MidpointRule( a, b, model, n):
   """estimate the integral from a to b of a likelihood function defined
      by a model.  The model provides the negative of the log likelihood
      function.  The function integrated is f(x) = e^{model(x)}.
   """

   h   = 1/n          # step size for the numerical integration
   sum = 0.           # multiply by h at the end
   for k in range(n):
      xk   = (k+.5)*h              # midpoint of interval k
      sum += np.exp( - model(xk))   # integrate the likelihood, which is the
                                   # exponential of the log likelihood.
   return h*sum
```

Figure 6: A module with code for the midpoint rule integration.

Figure 7 puts everything together. Line 7 gets the data model, which will be a log likelihood function. Line 8 gets the integration algorithm, the midpoint rule in this case. Lines 14 to 21 create fake "data" to illustrate working with data. In a real application, the data probably would come from a data file. Some future Python note for this course will discuss reading from a data file. That said, it is common to test computational methods on fake data before running them on real data, so models with fake data happen in real computing projects. Line 25 *instantiates* (makes an instance, creates) the LogLikelihood object and binds the name `model1` to it. Lines 32 and 33 add new names to the namespace of the `model1` object. These allow the `slice` member function to work. You cannot (as far as I know) do anything like this in C++, where the members of a class are defined "at compile time" by the code defining the class. In Python, it is possible to have a class with a trivial constructor (no `__init__` and have all the data and functions added afterwards, "from the outside". That would make it hard for the "user" to figure out what the class is about.



```python
#  Demonstration Python 3 module for Week 3
#  Scientific Computing, Fall 2021, goodman@cims.nyu.edu
#  Illustrate classes, "this", and constructors
#  This module uses the class

import numpy as np
import LogLikelihood as LL    # LL for "log likelihood"
import MidpointRule as mr      # mr for "Midpoint Rule"

print("Getting classy")

# create fake data "observations"

n      =  4        # number of fake "data points"
omega  =  2.345    # arbitrary "physical" parameters
A      =  .4321
times  = np.zeros([n])    # allocate a numpy array, not a Python list
values = np.zeros([n])
for i in range(n):
   times[i]  = i
   values[i] = A*np.sin(omega*times[i])

#    Instantiate the model, using the fake data

model1   = LL.LL( times, values)

#   Set up for a computational experiment

op = 2.456     # op = "omega prime", a frequency not used to make the fake data
Ap = .543      # Ap = "A prime", a different amplitude

model1.omega = op    # insert "omega" into the namespace of model1 with value op
model1.t0    = .01

v1 = model1.slice(Ap)         # evaluate with the internal values of omega and t0
v2 = model1.ssq(op, Ap, .01)  # evaluate with all parameters given
print("v1 is " + str(v1) + ", v2 is " + str(v2) + ", difference is " + str(v2-v1))

#  Do the integration

a = 0.    #  range of integration, from a to b
b = 2.
n = 10     # number of integration points

I1 = mr.MidpointRule( a, b, model1.slice, n)
I2 = mr.MidpointRule( a, b, model1.slice, 2*n)
print("I1 is " + str(I1) + ", I2 is " + str(I2) + ", the difference is " + str(I2-I1))
```

Figure 7: Use a class object to define a function to be integrated.