

Advanced python features - part II

Computing Methods for Experimental Physics and Data Analysis

A. Manfreda

alberto.manfreda@pi.infn.it

INFN-Pisa

Compiled on October 28, 2019



Functions inside functions

- ▷ Function in python are **first class object**
- ▷ The name is a bit misleading, but what actually means is that functions can be passed as argument to other functions and returned as result from other functions
- ▷ This shouldn't surprise you much: functions are objects of a '*function*' class, so they behave like any other variable in Python
- ▷ Another thing you can (and sometimes want to) do is *defining* a function inside another.
- ▷ Let's see how it works



Functions inside functions

https://github.com/lucabaldini/cmepda/tree/master/slides/latex/snippets/inner_outer.py

```
1  def outer():
2      def inner(): # Defining the inner function inside the outer function
3          print('Inner function')
4          return # End of the inner function
5      return inner # Inner is the output of outer
6
7  my_func = outer() # my_func is now referncing 'inner'
8  print(my_func.__name__)
9  my_func() # Calling my_func is equal to calling 'inner'
10
11 def outer2():
12     some_string = 'Hello!'
13     def inner():
14         # We have access to the variables in the outer function!
15         print(some_string)
16     return inner
17
18 my_other_func = outer2()
19 my_other_func()
20
21 [Output]
22 inner
23 Inner function
24 Hello!
```



Closures and free variables

- ▷ When a function is created inside another function it has access to the local variables of the outer function, even after its scope ended
- ▷ This is technically possible because those variables are kept in a special space of memory, the **closure** of the inner function
- ▷ Such variables are called **free variables**
- ▷ Note: if you *assign* to a free variable in the inner function, by default a new, local variable is created instead!
- ▷ To avoid this you have to explicitly declare that you want to access the variable in the closure using the **nonlocal** keyword
- ▷ Remember: *'Explicit is better than implicit'*



Free variables - a mistake to avoid

https://github.com/lucabaldini/cmepda/tree/master/slides/latex/snippets/closure_wrong.py

```
1  def running_average():
2      total_count = 0
3      num_elements = 0
4      def accumulator(value):
5          total_count += value # Doesn't work! total_count is reassigned!
6          num_elements += 1 # Doesn't work! total_count is reassigned!
7          return total_count/num_elements
8      return accumulator
9
10 run_avg = running_average()
11 print(run_avg(1.))
12 print(run_avg(5.))
13 print(run_avg(2.5))
14
15 [Output]
16 Traceback (most recent call last):
17   File "snippets/closure_wrong.py", line 11, in <module>
18     print(run_avg(1.))
19   File "snippets/closure_wrong.py", line 5, in accumulator
20     total_count += value # Doesn't work! total_count is reassigned!
21 UnboundLocalError: local variable 'total_count' referenced before assignment
```



Free variables - the correct way

https://github.com/lucabaldini/cmepda/tree/master/slides/latex/snippets/closure_right.py

```
1  def running_average():
2      total_count = 0
3      num_elements = 0
4      def accumulator(value):
5          # We declare the relevant variables as nonlocal
6          nonlocal total_count, num_elements
7          # Now we can assign to them - the variables in the closure will be
8          # modified, as we want!
9          total_count += value
10         num_elements += 1
11         return total_count/num_elements
12     return accumulator
13
14 run_avg = running_average()
15 print(run_avg(1.))
16 print(run_avg(5.))
17 print(run_avg(2.5))
18
19 [Output]
20 1.0
21 3.0
22 2.8333333333333335
```



Wrapping functions

- ▷ The typical use of defining a function inside a function is to create a **wrapper**
- ▷ A wrapper is a function that calls another one adding a layer of functionalities in between - for example it may do some pre-process of the input, or change the output in some way, or measure the execution time or whatever we want
- ▷ The technique for creating a wrapper function in Python is:
 - ▷ Pass the function that we want to wrap as argument of the outer function
 - ▷ Inside the outer function we define an inner function, which is the actual wrapper
 - ▷ The wrapper calls the wrapped function and adds its functionalities, before and/or after the call. It may return the same output or a manipulated one.
 - ▷ Then from the outer function we return the wrapper



Wrapper

<https://github.com/lucabaldini/cmepda/tree/master/slides/latex/snippets/wrapper.py>

```
1 def some_function(a, b):
2     print('Executing {} x {}'.format(a, b))
3     return a * b
4
5 def add_n_wrapper(func, n): # We take the wrapped function as argument
6     """ This wrapper adds n to the result of the wrapped function """
7
8     def wrapper(*args, **kwargs):
9         """We pass the arguments as *arg, **kwargs, because this is the most
10         general form in Python: we can collect any combination of arguments like
11         that. Note that we have access to both 'func' and 'n', as they are stored
12         in the closure of 'wrapper' """
13         result = func(*args, **kwargs) # Pass the arguments to the wrapped function
14         print('Adding {}'.format(n))
15         return result + n # Return a modified result in this case
16
17     return wrapper # From add_n_wrapper we return the wrapper
18
19 function_plus_five = add_n_wrapper(some_function, 5)
20 print('Result = {}'.format(function_plus_five(2, 3)))
21
22 [Output]
23 Executing 2 x 3
24 Adding 5
25 Result = 11
```




Decorators

- ▷ Often, when you wrap a function, you don't want to change it's name, so you reassign the wrapped funtion to its old name
- ▷ In fact, this techinque is so common that python introduced a special syntax for it: decorators
- ▷ A decorated function has simply the name of the wrapper added with a '@' on top of its declaration



Decorators

<https://github.com/lucabaldini/cmepda/tree/master/slides/latex/snippets/decorator.py>

```
1 def print_function_info(func):
2     def wrapper(*args, **kwargs):
3         print('Calling function \'{}\'' .format(func.__name__))
4         print('Positional arguments = {}'.format(args))
5         print('Keyword arguments = {}'.format(kwargs))
6         return func(*args, **kwargs)
7     return wrapper
8
9 @print_function_info
10 def some_function(a, b, c=0):
11     return a * b + c
12
13 # This is equivalent to: some_function = print_function_info(some_function)
14
15 print(some_function(1, 2, c=7))
16 # Inspecting the function reveals that we are calling the wrapper
17 print('The name of the function is \'{}\'' .format(some_function.__name__))
18
19 [Output]
20 Calling function 'some_function'
21 Positional arguments = (1, 2)
22 Keyword arguments = 'c': 7
23 9
24 The name of the function is 'wrapper'
```



A decorator to measure execution time

https://github.com/lucabaldini/cmepda/tree/master/slides/latex/snippets/time_measuring_decor.py

```
1  import time
2  from functools import wraps
3
4  def clocked(func):
5      """ We use functools.wraps to keep the original function name and docstring"""
6      @wraps(func)
7      def wrapper(*args, **kwargs):
8          tstart = time.clock()
9          result = func(*args, **kwargs)
10         exec_time = time.clock() - tstart
11         print('Function {} executed in {} s'.format(func.__name__, exec_time))
12         return result
13     return wrapper
14
15 @clocked
16 def square_list(input_list):
17     """ Return the square of a list"""
18     return [item**2 for item in input_list]
19
20 # Make sure the function name and docstring look the same
21 print('\n{}\n': {}'.format(square_list.__name__, square_list.__doc__))
22 square_list(range(2000000))
23
24 [Output]
25 'square_list': Return the square of a list
26 Function square_list executed in 0.372302 s
```



The @classmethod decorator

- ▷ We have already seen a built-in Python decorator: *@property*
- ▷ We used that to get proper encapsulation
- ▷ There is another built-in decorator one which is very useful for classes: *@classmethod*
- ▷ A classmethod is like a class attribute: you don't need an instance to use it
- ▷ A class method can access class attributes but not instance attributes
- ▷ The main use for class methods is to provide *alternate constructors*



Class method

<https://github.com/lucabaldini/cmepda/tree/master/slides/latex/snippets/classmethod.py>

```
1  import numpy
2
3  class LabData:
4
5      def __init__(self, times, values):
6          """ Our usual constructor """
7          self.times = times
8          self.values = values
9
10     @classmethod # The classmethod decorator
11     def from_file(cls, file_path): # We get the class as first argument, not self
12         """ Constructor from a file """
13         print(cls)
14         times, values = numpy.loadtxt(file_path, unpack=True)
15         # We call the constructor of 'cls' which is our LabData
16         # This is not a 'real' constructor, we need to return the object!
17         return cls(times, values)
18
19     # We call the alternate constructor from the class itself, not from an instance!
20     lab_data = LabData.from_file('measurements.txt')
21     print(lab_data.values)
22
23     [Output]
24     <class '__main__.LabData'>
25     [15.2 12.4 11.7 13.2]
```



Context manager

- ▷ A **context manager** in Python is any class implementing the `__enter__` and `__exit__` method
- ▷ It is used with the syntax *with expression (as alias):*
- ▷ We have already seen context managers at work in opening files
- ▷ Their most important use is to make sure that all resources are correctly released **even when an exception is raised**
- ▷ However we can do other things with them



Measuring time with a context manager

https://github.com/lucabaldini/cmepda/tree/master/slides/latex/snippets/context_manager.py

```
1  class Clocking:
2      """ Context manager for time measurment. """
3      def __enter__(self):
4          import time
5          self.start_time = time.clock()
6          self.elapsed_time = 0.
7          return self # What you return here is assigned to 'as'
8
9      def __exit__(self, exc_type, exc_value, traceback):
10         """ Exit method. It gets notice of any exception raised in the body of the
11         with block. If no exception is raised, the arguments are all set to None """
12         import time
13         self.elapsed_time = time.clock() - self.start_time # Update the time
14         print('Exception type: {}, exception value: {}'.format(exc_type, exc_value))
15         # If you do nothing, any exception will propagate to the rest of the code.
16         # To stop that from happening you have to return True -- though you should
17         # do that only if it actually make sense to manage the exception here!
18         if exc_type is not None:
19             print('Exception handled succesfully.')
20             return True
21
22     with Clocking() as clock:
23         squares_list = [n**2 for n in range(1000000)]
24         raise RuntimeError('Let\'s see what happens!')
25     print('With block runned in {:.8f} seconds'.format(clock.elapsed_time))
26
27     [Output]
28     Exception type: <class 'RuntimeError'>, exception value: Let's see what happens!
29     Exception handled succesfully.
30     With block runned in 0.19369800 seconds
```



A recap exercise

- ▷ As an exercise to recap the previous lessons, we want to write a small class for representing a sequence of measurements
- ▷ In this case they are voltages taken at different times
- ▷ The features that our class needs to have are summarized by the following test module (not a real unittest):







Test of functionalities

https://github.com/lucabaldini/cmepda/tree/master/slides/latex/snippets/test_voltage_data.py

```
1  from voltage_data import VoltageData
2  from matplotlib import pyplot as plt
3
4  def run_tests(): # This is not a proper unittest module!
5      # Test constructor from data file
6      data_file = VoltageData.from_file('snippets/data/sample_data_file.txt')
7      # Column access by simple name
8      t = data_file.timestamps
9      v = data_file.voltages
10     print(t[0], v[0])
11
12     # Iterable by row
13     for row in data_file:
14         pass
15
16     # Proper representation and printing
17     print(repr(data_file))
18     print(data_file)
19
20     # Item access with slicing
21     print(data_file[1:5, :])
22
23     # Constructor from iterables (list, tuple, array)
24     data_file_2 = VoltageData(list(t), tuple(v))
25     # Check that the first row is the same
26     assert((data_file_2[0] == data_file[0]).all())
27     # Plotting
28     data_file_2.plot()
29     plt.show()
```



File we want to read

```
Open ▾  sample_data_file.txt Save     
~/.computing/cmepda/slides/latex  
1# Time[s] V[mV]  
2 0.1 0.24  
3 0.2 0.39  
4 0.3 0.58  
5 0.4 0.77  
6 0.5 0.84  
7 0.6 0.95  
8 0.7 1.00  
9 0.8 1.02  
10 0.9 0.95  
11 1.0 0.94  
12 1.1 0.83  
13 1.2 0.87  
14 1.3 0.48  
15 1.4 0.21  
16 1.5 0.04  
  
Plain Text ▾ Tab Width: 2 ▾ Ln 8, Col 4 ▾ INS
```



Solution - part 1

https://github.com/lucabaldini/cmepda/tree/master/slides/latex/snippets/voltage_data_1.py

```
1  import numpy
2
3  class VoltageData:
4      """Class for handling a set of measurements of the voltage at different
5      times."""
6
7      def __init__(self, times, voltages):
8          """ Constructor from two iterables (times and voltages) """
9          t = numpy.array(times, dtype=numpy.float)
10         v = numpy.array(voltages, dtype=numpy.float)
11         # Put together the arrays in a single matrix with column_stack
12         self._data = numpy.column_stack((t,v))
13
14     @classmethod
15     def from_file(cls, file_path):
16         """ Alternate constructor from a data file, exploiting load_txt() """
17         t, v = numpy.loadtxt(file_path, unpack=True)
18         return cls(t, v)
19
20     @property
21     def timestamps(self):
22         # Use the slice syntax to select the first column
23         return self._data[:, 0]
24
25     @property
26     def voltages(self):
27         # Use the slice syntax to select the second column
28         return self._data[:, 1]
```

Solution - part 2

https://github.com/lucabaldini/cmepda/tree/master/slides/latex/snippets/voltage_data_2.py

```

1  class VoltageData:
2
3      """ Other methods here... """
4
5  def __len__(self):
6      """ Number of data points (or columns in the file, which is the same) """
7      return self._data.shape[0]
8
9  def __getitem__(self, index):
10     # We use composition and simply call __getitem__ from _data
11     return self._data[index]
12
13  def __iter__(self):
14     """Return the values row by row"""
15     # We use a generator expression here. The syntax is very readable!
16     for i in range(len(self)):
17         yield self._data[i, :]
18
19  def __repr__(self):
20     """ Print the full content row by row """
21     return '\n'.join('{} {}'.format(row[0], row[1]) for row in self)
22
23  def __str__(self):
24     """ Print the full content row-by-row with a nice formatting """
25     row_fmt = 'Row {} -> {:.3f} s, {:.1f} mV'
26     row_str_gen = \
27         (row_fmt.format(i, row[0], row[1]) for i, row in enumerate(self))
28     return '\n'.join(row_str_gen)

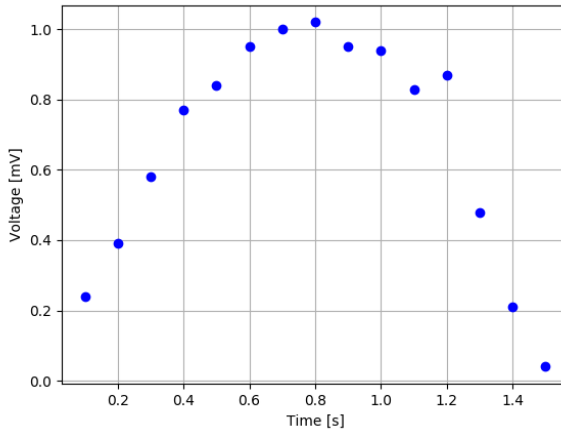
```

Solution - part 3

https://github.com/lucabaldini/cmepda/tree/master/slides/latex/snippets/voltage_data_3.py

```
1  class VoltageData:
2
3      """Other methods here..."""
4
5  def plot(self, ax=None, fmt='bo'):
6      """ Draw the data points."""
7      from matplotlib import pyplot as plt
8      # The user can provide an existing figure to add the plot, otherwise we
9      # create a new one.
10     if ax is not None:
11         plt.sca(ax) # sca (Set Current Axes) selects the given figure
12     else:
13         ax = plt.figure('voltage_vs_time')
14     plt.plot(self.timestamps, self.voltages, fmt)
15     plt.xlabel('Time [s]')
16     plt.ylabel('Voltage [mV]')
17     plt.grid(True)
18     return ax # We return the axes, just in case
```

Output plot





Put yourself to test

- ▷ Manage a third column - optional - with voltage errors
- ▷ Write a proper unittest module for the class
- ▷ The full current version of the class is in `snippets/voltage_data.py`