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 vairable 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

```python
def outer():
    def inner(): # Defining the inner function inside the outer function
        print(’Inner function’)
        return # End of the inner function
    return inner # Inner is the output of outer

my_func = outer() # my_func is now referncing ’inner’
print(my_func.__name__)
my_func() # Calling my_func is equal to calling ’inner’

def outer2():
    some_string = ’Hello!’
    def inner():
        # We have access to the variables in the outer function!
        print(some_string)
    return inner

my_other_func = outer2()
my_other_func()

[Output]
inner
Inner function
Hello!
```

Colusers 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 techincally possible because those varibales 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 explictly declare that you want to access the variable in the closure using the nonlocal keyword
- Remember: *'Explicit is better then implicit'*

Free variables - a mistake to avoid

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

```
def running_average():
    total_count = 0
    num_elements = 0
    def accumulator(value):
        total_count += value # Doesn’t work! total_count is reassigned!
        num_elements += 1 # Doesn’t work! total_count is reassigned!
        return total_count/num_elements
    return accumulator

run_avg = running_average()
print(run_avg(1.))
print(run_avg(5.))
print(run_avg(2.5))

[Output]
Traceback (most recent call last):
  File "snippets/closure_wrong.py", line 11, in <module>
    print(run_avg(1.))
  File "snippets/closure_wrong.py", line 5, in accumulator
    total_count += value # Doesn’t work! total_count is reassigned!
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

```python
def running_average():
    total_count = 0
    num_elements = 0
    def accumulator(value):
        # We declare the relevant variables as nonlocal
        nonlocal total_count, num_elements
        # Now we can assign to them - the variables in the closure will be
        # modified, as we want!
        total_count += value
        num_elements += 1
        return total_count/num_elements
    return accumulator

run_avg = running_average()
print(run_avg(1.))
print(run_avg(5.))
print(run_avg(2.5))

[Output]
1.0
3.0
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 techinque for creating a wrapper fucntion 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 fucntion we return the wrapper

Wrapper

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

```
def some_function(a, b):
    print(’Executing {} x {}’.format(a, b))
    return a * b

def add_n_wrapper(func, n): # We take the wrapped function as argument
    """ This wrapper adds n to the result of the wrapped function"""

    def wrapper(*args, **kwargs):
       """We passs the arguments as *arg, **kwargs, because this is the most
       general form in Python: we can collect any comination of arguments like
       that. Note that we have access to both ’func’ and ’n’, as they are stored
       in the closure of ’wrapper’"""
       result = func(*args, **kwargs) # Pass the arguments to the wrapped fucntion
       print(’Adding {}’.format(n))
       return result + n # Return a modified result in this case

    return wrapper # From add_n_wrapper we return the wrapper

function_plus_five = add_n_wrapper(some_function, 5)
print(’Result = {}’.format(function_plus_five(2, 3)))

[Output]
Executing 2 x 3
Adding 5
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

```python
def print_function_info(func):
    def wrapper(*args, **kwargs):
        print('Calling function \'{}\''.format(func.__name__))
        print('Positional arguments = {}'.format(args))
        print('Keyword arguments = {}'.format(kwargs))
        return func(*args, **kwargs)
    return wrapper

@print_function_info
def some_function(a, b, c=0):
    return a * b + c

# This is equivalent to: some_function = print_function_info(some_function)

print(some_function(1, 2, c=7))
 # Inspecting the function reveals that we are calling the wrapper
print('The name of the function is \'{}\''.format(some_function.__name__))

[Output]
Calling function 'some_function'
Positional arguments = (1, 2)
Keyword arguments = 'c': 7
9
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

```python
import time
from functools import wraps

def clocked(func):
    """ We use functools.wraps to keep the original function name and docstring"""
    @wraps(func)
    def wrapper(*args, **kwargs):
        tstart = time.clock()
        result = func(*args, **kwargs)
        exec_time = time.clock() - tstart
        print('Function {} executed in {} s'.format(func.__name__, exec_time))
        return result
    return wrapper

@clocked
def square_list(input_list):
    """ Return the square of a list"""
    return [item**2 for item in input_list]

# Make sure the function name and docstring look the same
print('\'{}\': {}'.format(square_list.__name__, square_list.__doc__))
square_list(range(2000000))

[Output]
'square_list':  Return the square of a list
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

```python
import numpy

class LabData:

  def __init__(self, times, values):
     """ Our usual constructor"""
     self.times = numpy.array(times, dtype=numpy.float64)
     self.values = numpy.array(values, dtype=numpy.float64)

  @classmethod # The classmethod decorator
  def from_file(cls, file_path): # We get the class as first argument, not self
      """ Constructor from a file"""
      print(cls)
      times, values = numpy.loadtxt(file_path, unpack=True)
      # We call the constructor of 'cls' which is our LabData
      # This is not a 'real' constructor, we need to return the object!
      return cls(times, values)

# We call the alternate constructor from the class itself, not from an instance!
lab_data = LabData.from_file('snippets/data/measurements.txt')
print(lab_data.values)

[Output]
<class '__main__.LabData'>
[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 excpetion 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

```
class Clocking:
    """ Context manager for time measurment."""
    def __enter__(self):
        import time
        self.start_time = time.clock()
        self.elapsed_time = 0.
        return self # What you return here is assigned to ’as’

    def __exit__(self, exc_type, exc_value, traceback):
        """ Exit method. It gets notice of any exception raised in the body of the
        with block. If no exception is raised, the arguments are all set to None """
        import time
        self.elapsed_time = time.clock() - self.start_time # Update the time
        print(’Exception type: {}, exception value: {}’.format(exc_type, exc_value))
        # If you do nothing, any exception will propagate to the rest of the code.
        # To stop that from happening you have to return True -- though you should
        # do that only if it actually make sense to manage the exception here!
        if exc_type is not None:
            print(’Exception handled succesfully.’)
            return True

with Clocking() as clock:
    squares_list = [n**2 for n in range(1000000)]
    raise RuntimeError(’Let\’s see what happens!’)
print(’With block runned in {:.8f} seconds’.format(clock.elapsed_time))

[Output]
Exception type: <class ’RuntimeError’>, exception value: Let’s see what happens!
Exception handled succesfully.
With block runned in 0.19369800 seconds
```

A recap excercise

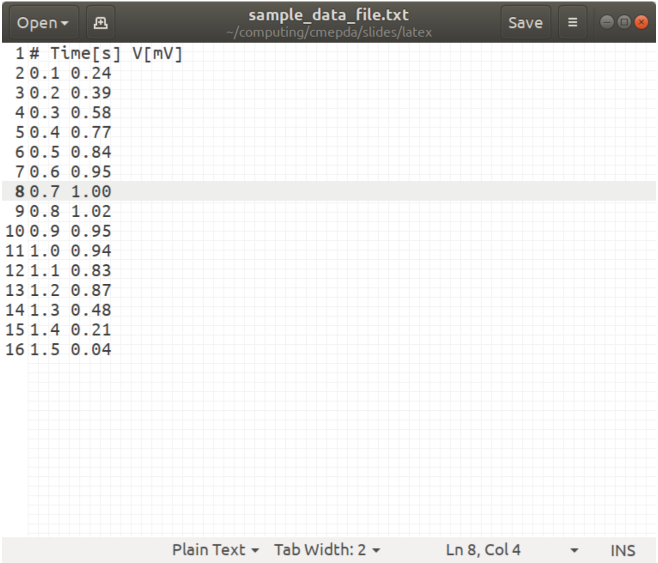
- As an exercise to recap the previous lessons, we want to write a small class for representing a sequence of measurements
- In this case they are voltages taken at different times
- The features that our class needs to have are summarized by the following test module (not a real unittest):

Test of functionalities

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

```python
from voltage_data import VoltageData
from matplotlib import pyplot as plt

def run_tests(): # This is not a proper unittest module!
    # Test constructor from data file
    data_file = VoltageData.from_file(’snippets/data/sample_data_file.txt’)
    # Column access by simple name
    t = data_file.timestamps
    v = data_file.voltages
    print(t[0], v[0])

    # Iterable by row
    for row in data_file:
        pass

    # Proper representation and printing
    print(repr(data_file))
    print(data_file)

    # Item access with slicing
    print(data_file[1:5, :])

    # Constructor from iterables (list, tuple, array)
    data_file_2 = VoltageData(list(t), tuple(v))
    # Check that the forst row is the same
    assert((data_file_2[0] == data_file[0]).all())
    # Plotting
    data_file_2.plot()
    plt.show()
```

File we want to read



```
# Time[s] V[mV]
0.1 0.24
0.2 0.39
0.3 0.58
0.4 0.77
0.5 0.84
0.6 0.95
0.7 1.00
0.8 1.02
0.9 0.95
1.0 0.94
1.1 0.83
1.2 0.87
1.3 0.48
1.4 0.21
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

```python
import numpy

class VoltageData:
    """Class for handling a set of measurements of the voltage at different
    times."""

    def __init__(self,  times, voltages):
        """ Constructor from two iterables (times and voltages)"""
        t = numpy.array(times, dtype=numpy.float64)
        v = numpy.array(voltages, dtype=numpy.float64)
        # Put together the arrays in a single matrix with column_stack
        self._data = numpy.column_stack((t,v))

    @classmethod
    def from_file(cls, file_path):
        """ Alternate constructor from a data file, exploiting load_txt()"""
        t, v = numpy.loadtxt(file_path, unpack=True)
        return cls(t, v)

    @property
    def timestamps(self):
        # Use the slice syntax to select the first column
        return self._data[:, 0]

    @property
    def voltages(self):
        # Use the slice syntax to select the second column
        return self._data[:, 1]
```

Solution - part 2

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

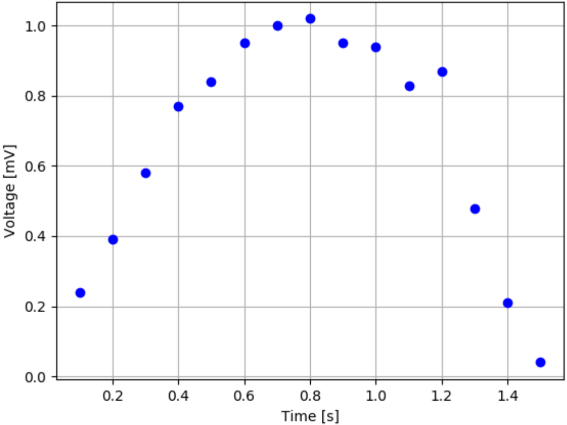
```python
class VoltageData:

    """ Other methods here..."""

    def __len__(self):
        """ Number of data points (or rows in the file, which is the same) """
        return self._data.shape[0]

    def __getitem__(self, index):
        # We use composition and simply call __getitem__ from _data
        return self._data[index]

    def __iter__(self):
        """Return the values row by row"""
        # We use a generator expression here. The syntax is very readible!
        for i in range(len(self)):
            yield self._data[i, :]

    def __repr__(self):
        """ Print the full content row by row """
        return '\n'.join('{} {}'.format(row[0], row[1]) for row in self)

    def __str__(self):
        """ Print the full content row-by-row with a nice formatting"""
        row_fmt = 'Row {} -> {:.1f} s, {:.2f} mV'
        row_str_gen = \
                (row_fmt.format(i, row[0], row[1]) for i, row in enumerate(self))
        return '\n'.join(row_str_gen)
```

Solution - part 3

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

```python
class VoltageData:

    """Other methods here..."""

    def plot(self, ax=None, fmt='bo'):
        """ Draw the data points."""
        from matplotlib import pyplot as plt
        # The user can provide an existing figure to add the plot, otherwise we
        # create a new one.
        if ax is not None:
            plt.sca(ax) # sca (Set Current Axes) selects the given figure
        else:
            ax = plt.figure('voltage_vs_time')
        plt.plot(self.timestamps, self.voltages, fmt)
        plt.xlabel('Time [s]')
        plt.ylabel('Voltage [mV]')
        plt.grid(True)
        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