Programming in Python

MMM001 - Data Engineering

https://github.com/chbrandt/MMM001

Carlos H Brandt

c.brandt@jacobs-university.de

Table of Contents

- Debugging
- Profiling
- Modules & Packages
  - Package structure
- External packages
- Laboratory

Python debugger

- Pdb is distributed as part of the Standard Library
- As (most) other debuggers, pdb will instrument a code during its execution to
  - Allow the user to inspect the state of the code in-memory in different parts during execution
  - Provide the user with the state of the code in-memory at the moment of a crash (post-mortem)

The user can set breakpoints or "walk" through the running code statements/lines and check (*i.e.*, print) the code state (*i.e.*, variables).

Python debugger

- To use pdb we have to import the 'pdb' module and 'pdb.set_trace()' at the first breakpoint in our code:

```
import pdb
(...)
pdb.set_trace()
(...)
```

- Then, we use Pdb' commands to navigate and inspect the state of the code:
  - print / p
  - next / n
  - step / s
  - continue / c
  - list / l
  - longlist / ll
  - breakpoint / b
  - disable/enable #
  - help / h

Python debugger

- We can also run a code instrumented by 'pdb' from the command line,

```
$ python -m pdb myprog.py
```

See pdb docs and RealPython for an extensive tutorial.

Python debugger

- For example,

```
num_list = [500, 600, 700]
alpha_list = ['x', 'y', 'z']
def nested_loop():
    for number in num_list:
        print(number)
        for letter in alpha_list:
            print(letter)
if __name__ == '__main__':
    nested_loop()
```

Profiling

- To *profile* a code is to measure a set of resources consumed during execution; Notably, memory and time consumed by different parts of the program.
- Typically -- as it is the case of Python standard profilers --, a rich set of statistics is reported to support the analysis of where and how resources are being (mis)used.
- Oftenly, we are interested in the time-performance of our code or -- more precisely -- in our code's (parts) time-complexity. That can be simplistically accessed with Python's timeit module.

Modules & Packages

- A *module* is a file containing Python definitions and statements. Also called a *script*, a module may define a running program as well as define functions, classes, data for other modules to use.
  - Once imported, modules provides an attribute `'__name__'` with the name of the module itself.
  - If the module is run as *script*, `'__name__'` has the value `'__main__'`

- A *package* is a collection of modules arranged in one or more directories.
  - To make a directory "importable" -- and so define a basic *package* -- it is sufficient to add a file named `'__init__.py'` in it.
  - `'__init__.py'` may be an empty file as well as provide definitions as any other module.

Modules & Packages

```
mypkg/
    __init__.py
    suba/
        __init__.py
        mod.py
    subb/
        __init__.py
        amod.py
        bmod.py
```

```
>>> import mypkg
>>> dir(mypkg)

>>> import mypkg.suba
>>> dir(mypkg.suba)

>>> from mypkg.subb import amod
>>> dir(amod)
```

Modules & Packages

- Within the modules of a package, *imports* can be *absolute* or *relative*

- *Absolute*, 'bmod' from 'amod'

```
from mypkg.subb import amod
```

- *Relative*, 'bmod' from 'amod'

```
from . import amod
```

External packages

- Numerical analysis
  - Numpy
  - Scipy
- N-D data handling
  - Pandas
  - Xarray
  - Datashader
- Visualization
  - Matplotlib
  - Seaborn
  - Bokeh
- Database
  - Sqlite3
  - SQLAlchemy
  - Pymongo
- Web
  - Beautifulsoup4
  - Json
  - Lxml
  - Requests
- Graph
  - Networkx

External packages

- GIS
  - Geopandas
  - Cartopy
  - Shapely
  - Fiona
  - Geopy
- Astronomy
  - Astropy
- Biology
  - Biopython
- Image processing
  - Scikit-image
  - OpenCV
  - Pillow
- Machine Learning
  - Scikit-learn
  - Keras
  - Tensorflow
- Text analysis
  - NLTK

Laboratory

- We will now use our classic Birthday-paradox to exercise
    - Structuring modules/package
    - Time-profile code
    - Optimize
    - Plot *time* .vs. *data-size*
    - (probably) debug
- We will create a package with different versions of the birthday-paradox:
    - "vanilla": using pure built-in data structures (for numbers generation/computations)
    - "numpy": using numpy array (for numbers generation/computations)
- We will measure the (total) time necessary to <u>run the different versions</u> of the problem <u>for different data sizes</u> (#-people, #-simulations)
    - **For each version, a file with two columns should be created**: *data-size , run-time*

Laboratory

- Let's use the following structure

```
birthday_paradox/
    __init__.py
    vanilla/
        __init__.py
    numpy/
        __init__.py
    io/
        __init__.py
    plot/
        __init__.py
```

- In 'vanilla' the implementation of built-in python data functions
- In 'numpy' the implementation of numpy data functions
- In 'io' the functions to write the (CSV) file with data-size and time to run either the "vanilla" and "numpy" implementations
- In 'plot' the functions to plot the data-size .vs. time measurements