04.advanced

March 21, 2018

1 Avanced Python

1.1 Tips

- On all your script :
- shebang `#!/usr/bin/env python3`
- encoding `# -*- coding: utf-8 -*-`
- underscore have a lot of different meanings
- separator in number `10_000` (only python 3.6)
- last result in the interpreter `_`
- I don't care `_ = f()` (dangerous with internationnaization)
- weakly private `_something` (won't be imported with `import *`)
- avoid conflict `list_`
- more private (mangled) `__stuff` -> `_ClassName__mangled_stuff`
- magic methods (also mangled) `__init__`
- for internationnalization `_()`
- I/O
- **Always** decode in input

- **Always** encode in output
- modules : import involve execution!
  Use

  ```
  if __name__ == '__main__':
  some_computation()
  ```

- unpacking dans les boucles

- zip:

  ```
  for name, surname in zip(names, surnames):
  ...
  ```

- enumerate

  ```
  for index, prime in enumerate(primes):
  ...
  ```

- False tests :

  ```
  False, 0, None, __nonzero__(), __len__()
  ```

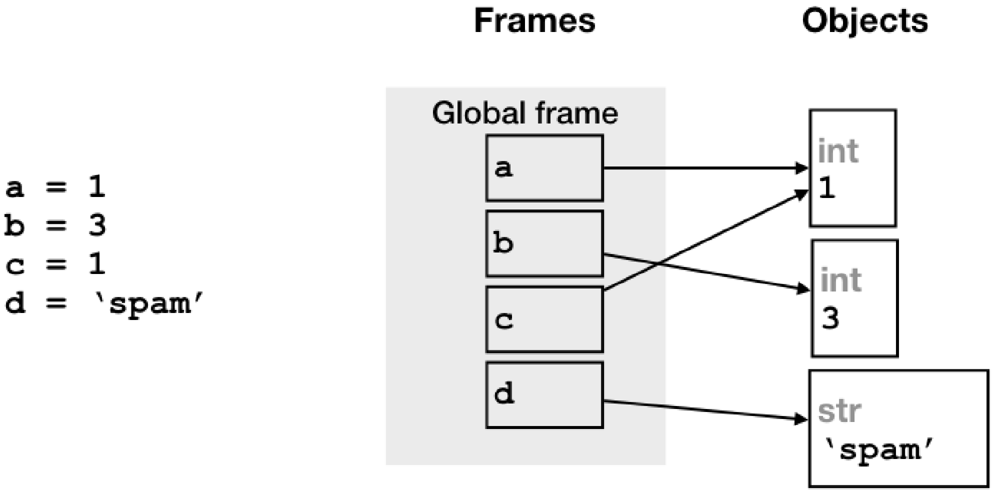
- lambda functions

```
In [1]: square1 = lambda x: x ** 2

        def square2(x):
            return x ** 2

        print(square1(5))
        print(square2(5))
```

```
25
25
```

- Be carefull with shared references

  ```
  a is b # a and b are references to the same object
  ```



```
a = 1
b = 3
c = 1
d = 'spam'
```

shared references image

```
In [2]: a = 2
        b = a
        print(a, b)
        a = 3
        print(a, b)
        l1 = [1, 2, 3]
        l2 = l1
        l3 = l1[:]
        print(l1, l2, l3)
        l1[1] = 4
        print(l1, l2, l3)
```

```
2 2
3 2
[1, 2, 3] [1, 2, 3] [1, 2, 3]
[1, 4, 3] [1, 4, 3] [1, 2, 3]
```

- **never** use a mutable optionnal arg

```
In [3]: def wrong(a_list=[]):  # never use a mutable optionnal argument
            a_list.append(1)
            return a_list
        print(wrong())
        print(wrong())
        print(wrong([2]))
        print(wrong())

        print()
        def good(a_list=None):
            if a_list is None:
                a_list = []
            a_list.append(1)
            return a_list
        print(good())
        print(good())
        print(good([2]))
        print(good())
```

```
[1]
[1, 1]
[2, 1]
[1, 1, 1]

[1]
[1]
[2, 1]
[1]
```

- use a shallow copy to modify a list in a loop (or be very carefull)

```
In [4]: from copy import copy

        def good(my_set):
            for value in copy(my_set):
                if 'bert' in value:
                    my_set.remove(value)
            return(my_set)

        def wrong(my_set):
            for value in my_set:
                if 'bert' in value:
                    my_set.remove(value)
            return(my_set)

        print("list  ok ", good(["einstein", "albert", "bert", "curie"]))
        print("set   ok ", good({"einstein", "albert", "bert", "curie"}))

        print("list Nok ", wrong(["einstein", "albert", "bert", "curie"]))
        print("set  Nok ", wrong({"einstein", "albert", "bert", "curie"}))
        print("END")
```

```
list  ok  ['einstein', 'curie']
set   ok  {'curie', 'einstein'}
list Nok  ['einstein', 'bert', 'curie']
```

```
        ---------------------------------------------------------------------------

        RuntimeError                              Traceback (most recent call last)

        <ipython-input-4-92da89fdbec5> in <module>()
         17
         18 print("list Nok ", wrong(["einstein", "albert", "bert", "curie"]))
    ---> 19 print("set  Nok ", wrong({"einstein", "albert", "bert", "curie"}))
         20 print("END")


        <ipython-input-4-92da89fdbec5> in wrong(my_set)
          8
          9 def wrong(my_set):
    ---> 10     for value in my_set:
         11         if 'bert' in value:
         12             my_set.remove(value)
```

```
RuntimeError: Set changed size during iteration
```

- use exceptions
  - try is very fast but except is very slow
  - nice way to get out of multiples loops/functions at the same time
  - Allows you to be sure that an error had been taken care of

```
In [5]: try:
            print("set  Nok ", wrong({"einstein", "albert", "bert", "curie"}))

        except RuntimeError as e:
            print()
            print("Oups, something went wrong:")
            print(e)
            print("Continuing anyway")
            print()

        print("I'm continuing")
```

```
Oups, something went wrong:
Set changed size during iteration
Continuing anyway

I'm continuing
```

```
In [6]: import sys
        class MyException(Exception):
            pass

        try:
            for i in range(10):
                for j in range(10):
                    if i == j == 5:
                        raise MyException("Found it")
        except MyException as e:
            print("Out of the loop")
            print(e)
            print("Stop")

            # In a script, use a non-zero return code
            # exit(1)

            # In jupyter you can do
            raise

        print("I will never appear")
```

```
Out of the loop
Found it
Stop
```

```
    ---------------------------------------------------------------------------

    MyException                               Traceback (most recent call last)

    <ipython-input-6-4fd9146a2d0c> in <module>()
      7         for j in range(10):
      8             if i == j == 5:
----> 9                 raise MyException("Found it")
     10 except MyException as e:
     11     print("Out of the loop")


    MyException: Found it
```

- Use decorator
- debugging, timing, ...

```
In [7]: from functools import wraps

        def PrintAppel(f):
            def before_f():
                new_f.NbAppels += 1
                print("Entering {}".format(f.__name__))

            def after_f():
                print("Exiting {}".format(f.__name__))
                print("This was the call nř {}".format(new_f.NbAppels))

            @wraps(f)
            def new_f(*args, **xargs):
                before_f()
                res = f(*args, **xargs)
                after_f()
                return res

            new_f.NbAppels = 0

            return new_f
```

```
In [8]: @PrintAppel
        def a_function(x):
            return 2 * x
```

```
In [9]: a_function(2)

Entering a_function
Exiting a_function
This was the call nř 1


Out[9]: 4
```

- Make use of classes in order to isolate your work (and your bugs)

```
In [10]: class egg(object):                              # All objects derived from the same obj
             """ Full exemple of a class in python """
             total_number = 0                            # shared attribut between all instances

             def __init__(self, number=1):               # constructor
                 """ constructor from number """
                 self.number = number                    # Good way of defining attributes
                 egg.total_number += number

             @classmethod
             def from_recipe(cls, recipe):               # Alternative constructor
                 """ constructor from recipe """
                 return cls(recipe["oeufs"])

             def __del__(self):                          # destructor (rare)
                 """ destructor """
                 egg.total_number -= self.number

             def __str__(self):                          # convert your object into printable st
                 """ egg to str convertor """
                 return "On a total of {} eggs, I own {}".format(egg.total_number, self.number)

             def how_many(self):                         # a function of the instance
                 """ Return the current number of eggs in the recipe """
                 return self.number

             @staticmethod
             def how_many_egg():                          # a function on the class (rare)
                 """ Return the total number of eggs for all recipes """
                 return egg.total_number

         if __name__ == "__main__":

             fried_egg = egg()
             omelette = egg(3)
             recipe_pancake = {"oeufs":2, "lait":0.5, "farine":300}
             pancake = egg.from_recipe(recipe_pancake)
             print("Fried egg    : ", fried_egg)
```

```
        print("Omelette     : ", omelette)
        print("Pancake      : ", pancake)
        print()
        print("{:<12} : {:>5} | {}".format("egg",
                                           "NaN",
                                           egg.how_many_egg()))
        print("{:<12} : {:>5} | {}".format("fried_egg",
                                           fried_egg.how_many(),
                                           fried_egg.how_many_egg()))
        print("{:<12} : {:>5} | {}".format("omelette",
                                           omelette.how_many(), omelette.how_many_egg()))
        print("{:<12} : {:>5} | {}".format("pancake",
                                           pancake.how_many(),
                                           pancake.how_many_egg()))
        print()
        del omelette
        print("{:<12} : {:>5} | {}".format("egg",
                                           "NaN",
                                           egg.how_many_egg()))
        print("{:<12} : {:>5} | {}".format("fried_egg",
                                           fried_egg.how_many(),
                                           fried_egg.how_many_egg()))
        print("{:<12} : {:>5} | {}".format("pancake",
                                           pancake.how_many(),
                                           pancake.how_many_egg()))
        del fried_egg
        del pancake

        print()
        help(egg)
```

```
Fried egg    :  On a total of 6 eggs, I own 1
Omelette     :  On a total of 6 eggs, I own 3
Pancake      :  On a total of 6 eggs, I own 2

egg          :   NaN | 6
fried_egg    :     1 | 6
omelette     :     3 | 6
pancake      :     2 | 6

egg          :   NaN | 3
fried_egg    :     1 | 3
pancake      :     2 | 3

Help on class egg in module __main__:

class egg(builtins.object)
 |  Full exemple of a class in python
```

```
|
|  Methods defined here:
|
|  __del__(self)
|      destructor
|
|  __init__(self, number=1)
|      constructor from number
|
|  __str__(self)
|      egg to str convertor
|
|  how_many(self)
|      Return the current number of eggs in the recipe
|
|  ----------------------------------------------------------------------
|  Class methods defined here:
|
|  from_recipe(recipe) from builtins.type
|      constructor from recipe
|
|  ----------------------------------------------------------------------
|  Static methods defined here:
|
|  how_many_egg()
|      Return the total number of eggs for all recipes
|
|  ----------------------------------------------------------------------
|  Data descriptors defined here:
|
|  __dict__
|      dictionary for instance variables (if defined)
|
|  __weakref__
|      list of weak references to the object (if defined)
|
|  ----------------------------------------------------------------------
|  Data and other attributes defined here:
|
|  total_number = 0
```

- For launching external program :
- If you don't care about the output of the program `python subprocess.check_call(["cmd", "arg1", "arg2"])`
- otherwise (remember to decode) `python data = subprocess.check_output(["cmd", "arg1", "arg2"]).decode('utf-8')`

```
In [11]: import subprocess
         import sys
         data = subprocess.check_output([sys.executable, "script.py", "Marc"]).decode('utf-8')
         print(data)
```

```
OH HI MARC
```

1.2 Packaging

- respect PEP (not only for prettyness)
- docstring (auto-documentation)
- All fonctions
- All classes
- All modules (`__init__.py`)
- All files
- type hinting (that's new)
- Almost totally ignored during execution
- mypy (and more and more IDE) are capable of checking consistency
- The typing module allows you to define complex types
- More and more package are complient with this

```
In [12]: def greeting(name: str) -> str:
             var = "Hello"   # type: str
             # python 3.7 : var = "Hello" : str

             return var + " " + name
```

- pytest (unit-testing)
- auto discovery (use tests folders, test_truc function, and TestMachin classes)
- allow parametrization

```
In [13]: #ONLY for ipython
         import ipytest.magics
         import pytest
         __file__ = '04.advanced.ipynb'
```

```
In [14]: %%run_pytest[clean] -qq
         #this was only for ipython

         def test_sorted():
             assert sorted([5, 1, 4, 2, 3]) == [1, 2, 3, 4, 5]

         # as does parametrize
         @pytest.mark.parametrize('input, expected', [
                                                     ([2, 1], [1, 2]),
                                                     ('zasdqw', list('adqswz')),
                                                     ]
```

```
                                )
          def test_exemples(input, expected):
              actual = sorted(input)
              assert actual == expected
```

```
...                                                                                                      [
```

- gettext (auto-internationnalization) ?
- argparse
- configParser
- logging
- print -> go to console (for ordinary usage)
- warning.warn -> go to console (usually once : for signaling a something the user should fix)
- logging.level -> go anywhere you want (for detailled output and/or diagnostic)

```
In [15]: import logging
         import warnings


         def prepare_logging():
             """
             Prepare all logging facilities

             This should be done in a separate module
             """

             # if not already done, initialize logging facilities
             logging.basicConfig()

             # create a logger for the current module
             logger = logging.getLogger(__name__)

             ## ONLY FOR IPYTHON
             # clean logger (ipython + multiple call)
             from copy import copy
             for handler in copy(logger.handlers):
                 logger.removeHandler(handler)
             # Do not propagate message to ipython (or else thy will be printed twice)
             logger.propagate=False
             ## ONLY FOR IPYTHON


             # optionnal : change format of the log
             logFormatter = logging.Formatter("%(asctime)s [%(threadName)-12.12s] [%(levelname)-

             # optionnal : create a handler for file output
             fileHandler = logging.FileHandler("{logPath}/{fileName}.log".format(logPath=".", fi
```

```python
    # optionnal : create a handler for console output
    consoleHandler = logging.StreamHandler()

    # optionnal : Apply formatter to both handles
    fileHandler.setFormatter(logFormatter)
    consoleHandler.setFormatter(logFormatter)

    # optionnal : attach handler to the logger
    logger.addHandler(fileHandler)
    logger.addHandler(consoleHandler)

    # what severity to log (default is NOTSET, i.e. all)
    logger.setLevel(logging.DEBUG)            # ALL
    fileHandler.setLevel(logging.INFO)        # NO DEBUG
    consoleHandler.setLevel(logging.WARNING)  # ONLY WARNING AND ERRORS

    return logger


def egg():
    warnings.warn("A warning only once")


if __name__ == "__main__":

    logger = prepare_logging()

    egg()

    logger.info('Start reading database')

    records = {'john': 55, 'tom': 66}

    logger.debug('Records: {}'.format(records))
    logger.info('Updating records ...')
    logger.warning("There is only 2 record !")
    logger.info('Saving records ...')
    logger.error("Something happend, impossible to save the records")
    logger.info('Restoring records ...')
    logger.critical("Database corrupted !")
    logger.info('End of program')

    egg()
```

```
04.advanced.ipynb:53: UserWarning: A warning only once
  "outputs": [
2018-03-21 13:43:53,927 [MainThread  ] [WARNI]  There is only 2 record !
2018-03-21 13:43:53,928 [MainThread  ] [ERROR]  Something happend, impossible to save the record
```

```
2018-03-21 13:43:53,929 [MainThread  ] [CRITI]  Database corrupted !
```

2 Performance

- profiling : Only optimize the bottlenecks !
- timeit (for small snippets of code)

```
In [16]: %timeit [1 + i for i in range(1,10000)]
         %timeit [1 * i for i in range(1,10000)]
         %timeit [1 / i for i in range(1,10000)]
         %timeit [1 // i for i in range(1,10000)]

         %timeit [1. + float(i) for i in range(1,10000)]
         %timeit [1. * float(i) for i in range(1,10000)]
         %timeit [1. / float(i) for i in range(1,10000)]
         %timeit [1. // float(i) for i in range(1,10000)]
```

```
509 ţs ś 3.22 ţs per loop (mean ś std. dev. of 7 runs, 1000 loops each)
465 ţs ś 5.18 ţs per loop (mean ś std. dev. of 7 runs, 1000 loops each)
481 ţs ś 1.15 ţs per loop (mean ś std. dev. of 7 runs, 1000 loops each)
398 ţs ś 4.64 ţs per loop (mean ś std. dev. of 7 runs, 1000 loops each)
1.57 ms ś 12.7 ţs per loop (mean ś std. dev. of 7 runs, 1000 loops each)
1.56 ms ś 5.31 ţs per loop (mean ś std. dev. of 7 runs, 1000 loops each)
1.59 ms ś 15.1 ţs per loop (mean ś std. dev. of 7 runs, 1000 loops each)
2.24 ms ś 40.4 ţs per loop (mean ś std. dev. of 7 runs, 100 loops each)
```

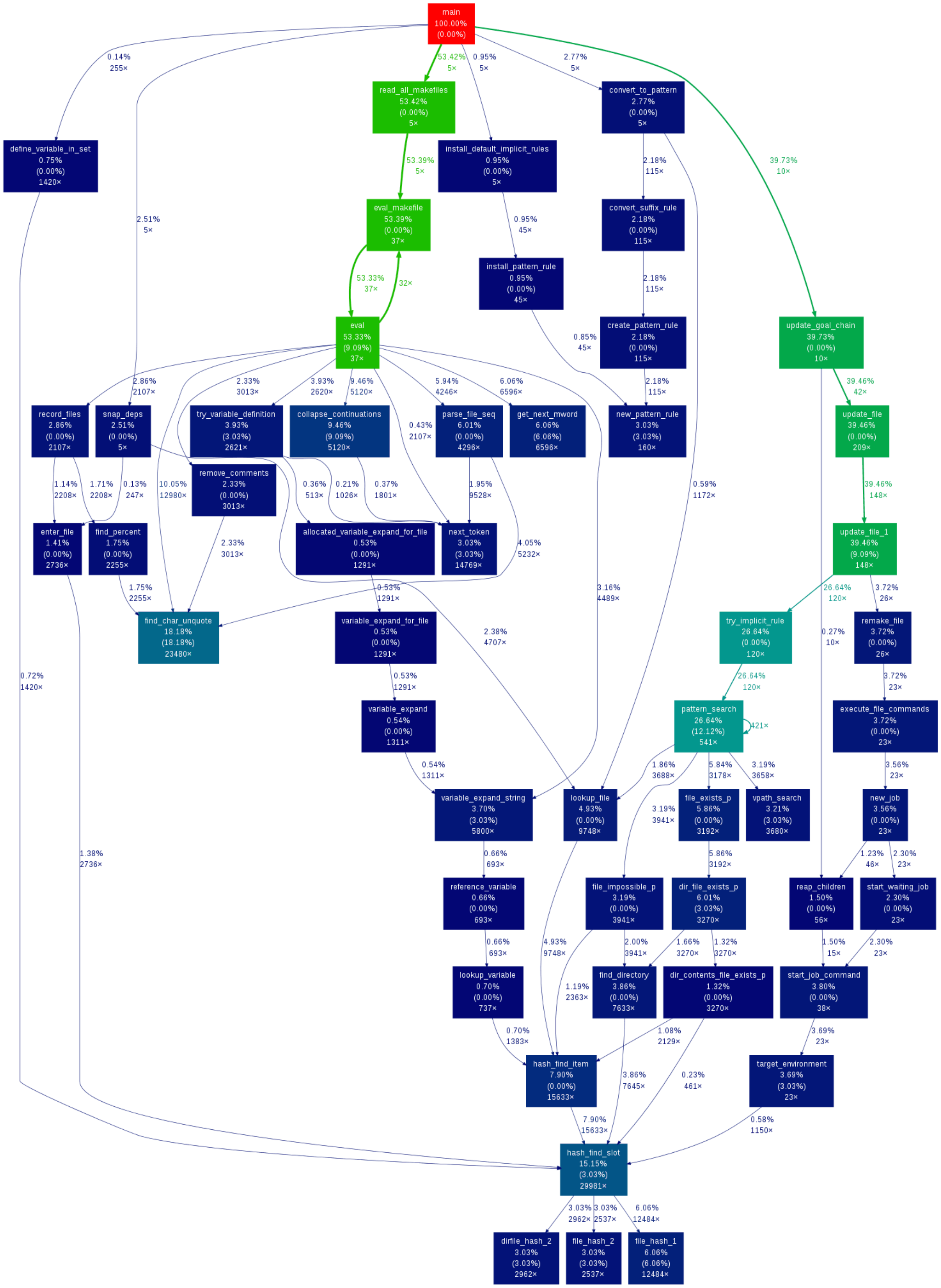
- cPfofile (for real code)
  ```
  python3 -m cProfile -o profile.pstats script.py
  gprof2dot -f pstats profile.pstats | dot -Tpng -o profile.png
  ```

```
In [17]: import numpy as np
         import cProfile
         import re

         def function2(array):
             for i in range(500):
                 array += 3
                 array = array * 2
             return array

         def function1():
             array = np.random.randint(500000, size=5000000)
             array = function2(array)
             return sorted(array)

         cProfile.run('function1()', sort="tottime")
```

```
       7 function calls in 5.105 seconds

 Ordered by: internal time

 ncalls  tottime  percall  cumtime  percall filename:lineno(function)
      1    4.786    4.786    4.786    4.786 <ipython-input-17-d5116138ace8>:5(function2)
      1    0.251    0.251    0.251    0.251 {built-in method builtins.sorted}
      1    0.039    0.039    5.105    5.105 <string>:1(<module>)
      1    0.028    0.028    0.028    0.028 {method 'randint' of 'mtrand.RandomState' objects}
      1    0.000    0.000    5.066    5.066 <ipython-input-17-d5116138ace8>:11(function1)
      1    0.000    0.000    5.105    5.105 {built-in method builtins.exec}
      1    0.000    0.000    0.000    0.000 {method 'disable' of '_lsprof.Profiler' objects}
```

```
In [18]: from IPython.display import Image
         Image(filename='images/profile.png')
```

```
Out[18]:
```



- in sequential and in Python

  - small is beautifull (PEP 20)
  - IO cost a lot (avoid reading and writing into files)
  - choose the right data structure / algorithm
  - prefere numpy based array
  - avoid loops (vectorization using slice)
  - avoid copy of array
  - changing size of an array > Stop optimizing your Python code here (and compile it)
  - inline manually
  - local is faster than global (and avoid dots)
  - use the out argument in numpy

- compiler
- Almost no modification to your code
  - numexpr (only small expression)
  - numba
- Almost need a total rewrite
  - cython
    * compilation must be done separately
  - f2py
    * included with numpy
    * compilation must be done separately
    * be carefull to the memory ordering

2.1 Python code (reference)

```
In [19]: import numpy as np

         def pyfunction1(a, b):
             return a * b - 4.1 * a > 2.5 * b

         def pyfunction2(a, b):
             return np.sin(a) + np.arcsinh(a / b)

         def convolve_python(f, g):
             # f is an image and is indexed by (v, w)
             # g is a filter kernel and is indexed by (s, t),
             #   it needs odd dimensions
             # h is the output image and is indexed by (x, y),

             if g.shape[0] % 2 != 1 or g.shape[1] % 2 != 1:
                 raise ValueError("Only odd dimensions on filter supported")

             # smid and tmid are number of pixels between the center pixel
             # and the edge, ie for a 5x5 filter they will be 2.
             vmax = f.shape[0]
             wmax = f.shape[1]
```

```
    smax = g.shape[0]
    tmax = g.shape[1]

    smid = smax // 2
    tmid = tmax // 2

    # Allocate result image.
    h = np.zeros_like(f)

    npsum = np.sum

    # Do convolution
    for x in range(smid, vmax - smid):
        for y in range(tmid, wmax - tmid):
            # Calculate pixel value for h at (x,y). Sum one component
            # for each pixel (s, t) of the filter g.

            #value = 0
            #for s in range(smax):
            #    for t in range(tmax):
            #        v = x - smid + s
            #        w = y - tmid + t
            #        value += g[s, t] * f[v, w]
            #h[x, y] = value

            # exploit vectorization

            v1 = x - smid
            v2 = v1 + smax
            w1 = y - tmid
            w2 = w1 + tmax
            h[x, y] = npsum(g * f[v1:v2, w1:w2])

    return h
```

In [20]:

```
def init_copy(f, g):
    return f.copy(), g.copy()

def finish(res):
    return res

def check(fun, data, ref=None, setup=init_copy, finish=finish, timing=True):

    print("Testing ", fun.__name__)

    f, g = setup(*data)

    if ref is None:
```

```
                 res = finish(fun(f, g))
             else:
                 res = finish(fun(f, g))

                 if type(ref) is not type(res):
                     print("types are differents : ",type(ref), type(res))
                     return res

                 if ref.dtype != res.dtype:
                     print("dtypes are differents : ",ref.dtype, res.dtype)
                     return res

                 if not np.array_equal(res, ref):
                     print("results are differents")
                     print("  ref shape: ", ref.shape)
                     print("  res shape: ", res.shape)
                     print()
                     print("  ref\n", ref)
                     print("  res\n", res)
                     #print()
                     #print("  ref\n", ref[4:-4, 4:-4])
                     #print("  res\n", res[4:-4, 4:-4])
                     return res

             if timing:
                 %timeit fun(f, g)

             return res

In [21]: # Create data
         a = np.arange(1,1e6)
         b = np.arange(1,1e6)

         n = 4
         s = 2 * n + 1
         g = np.arange(s ** 2, dtype=np.int).reshape((s, s))

         N = 200
         small_f = np.arange(N * N, dtype=np.int).reshape((N, N))
         N = 2000
         large_f = np.arange(N * N, dtype=np.int).reshape((N, N))

In [22]: # Initial result
         ref1 = check(pyfunction1, (a, b))
         ref2 = check(pyfunction2, (a, b))
         print()
         ref_small_convolve = check(convolve_python, (small_f, g))

Testing  pyfunction1
```

```
4.84 ms ś 407 ţs per loop (mean ś std. dev. of 7 runs, 100 loops each)
Testing  pyfunction2
60.1 ms ś 864 ţs per loop (mean ś std. dev. of 7 runs, 10 loops each)

Testing  convolve_python
167 ms ś 1.17 ms per loop (mean ś std. dev. of 7 runs, 10 loops each)
```

2.1.1 We can look at the bytecode (halfway to assembly)

And use it to optimize some things

```
In [23]: import dis
         print(dis.code_info(pyfunction2))
         print("Code :")
         dis.dis(pyfunction2)
         print()
```

```
Name:               pyfunction2
Filename:           <ipython-input-19-7d07a1a4a73c>
Argument count:     2
Kw-only arguments: 0
Number of locals:  2
Stack size:         4
Flags:              OPTIMIZED, NEWLOCALS, NOFREE
Constants:
   0: None
Names:
   0: np
   1: sin
   2: arcsinh
Variable names:
   0: a
   1: b
Code :
  7           0 LOAD_GLOBAL              0 (np)
              3 LOAD_ATTR                1 (sin)
              6 LOAD_FAST                0 (a)
              9 CALL_FUNCTION            1 (1 positional, 0 keyword pair)
             12 LOAD_GLOBAL              0 (np)
             15 LOAD_ATTR                2 (arcsinh)
             18 LOAD_FAST                0 (a)
             21 LOAD_FAST                1 (b)
             24 BINARY_TRUE_DIVIDE
             25 CALL_FUNCTION            1 (1 positional, 0 keyword pair)
             28 BINARY_ADD
             29 RETURN_VALUE
```

```
In [24]: from numpy import sin, arcsinh

         # We can avoid the step 0 and 12
         def pyfunction2_bis(a, b):
             return sin(a) + arcsinh(a / b)
```

```
In [25]: import dis
         print(dis.code_info(pyfunction2_bis))
         print("Code :")
         dis.dis(pyfunction2_bis)
         print()
```

```
Name:              pyfunction2_bis
Filename:          <ipython-input-24-d3f06e8d7add>
Argument count:    2
Kw-only arguments: 0
Number of locals:  2
Stack size:        4
Flags:             OPTIMIZED, NEWLOCALS, NOFREE
Constants:
   0: None
Names:
   0: sin
   1: arcsinh
Variable names:
   0: a
   1: b
Code :
  5           0 LOAD_GLOBAL              0 (sin)
              3 LOAD_FAST                0 (a)
              6 CALL_FUNCTION            1 (1 positional, 0 keyword pair)
              9 LOAD_GLOBAL              1 (arcsinh)
             12 LOAD_FAST                0 (a)
             15 LOAD_FAST                1 (b)
             18 BINARY_TRUE_DIVIDE
             19 CALL_FUNCTION            1 (1 positional, 0 keyword pair)
             22 BINARY_ADD
             23 RETURN_VALUE
```

```
In [26]: _ = check(pyfunction2_bis, (a, b), ref=ref2)
```

```
Testing  pyfunction2_bis
59.8 ms ś 203 ţs per loop (mean ś std. dev. of 7 runs, 10 loops each)
```

2.2 When possible use already available function

```
In [27]: from scipy.signal import convolve2d

         def scipy_convolve(f, g):

             vmax = f.shape[0]
             wmax = f.shape[1]

             smax = g.shape[0]
             tmax = g.shape[1]

             smid = smax // 2
             tmid = tmax // 2

             h = np.zeros_like(f)
             h[smid:vmax - smid, tmid:wmax - tmid] =  convolve2d(f, g, mode="valid")

             return h

         def scipy_setup(f, g):
             # for some reason, scipy take the filter in reverse...
             gr = g[::-1,::-1]
             return f.copy(), gr.copy()
```

```
In [28]: _ = check(scipy_convolve, (small_f, g), ref=ref_small_convolve, setup=scipy_setup)
         ref_large_convolve = check(scipy_convolve, (large_f, g), setup=scipy_setup)
```

```
Testing  scipy_convolve
6.16 ms ś 57.1 ţs per loop (mean ś std. dev. of 7 runs, 100 loops each)
Testing  scipy_convolve
659 ms ś 6.49 ms per loop (mean ś std. dev. of 7 runs, 1 loop each)
```

2.3 numexpr allows compilation of very simple code

And is multithreaded

```
In [29]: import numexpr as ne

         def ne_function1(a, b):
             return ne.evaluate('a * b - 4.1 * a > 2.5 * b')

         def ne_function2(a, b):
             return ne.evaluate("sin(a) + arcsinh(a / b)")
```

```
In [30]: _ = check(ne_function1, (a, b), ref=ref1)
         _ = check(ne_function2, (a, b), ref=ref2)
```

```
Testing  ne_function1
977 ţs ś 5.74 ţs per loop (mean ś std. dev. of 7 runs, 1000 loops each)
Testing  ne_function2
12.4 ms ś 616 ţs per loop (mean ś std. dev. of 7 runs, 100 loops each)
```

2.4 Numba allows compilation of more complex code using only decorators

And can parallelize part of your code
But it doesn't work everytime

```
In [31]: import numba as nb

         @nb.jit(nopython=True, nogil=True, cache=False, parallel=True)
         def nb_function1(a, b):
             return a * b - 4.1 * a > 2.5 * b

         @nb.jit(nopython=True, nogil=True, cache=False, parallel=True)
         def nb_function2(a, b):
             return np.sin(a) + np.arcsinh(a / b)

         @nb.jit(nopython=True, nogil=True, cache=False, parallel=True)
         def convolve_kernel(f, g):
             # smid and tmid are number of pixels between the center pixel
             # and the edge, ie for a 5x5 filter they will be 2.
             vmax = f.shape[0]
             wmax = f.shape[1]
             smax = g.shape[0]
             tmax = g.shape[1]

             smid = smax // 2
             tmid = tmax // 2

             # Allocate result image.
             h = np.zeros_like(f)

             # Do convolution
             for x in range(smid, vmax - smid):
                 for y in range(tmid, wmax - tmid):
                     # Calculate pixel value for h at (x,y). Sum one component
                     # for each pixel (s, t) of the filter g.

                     value = 0
                     for s in range(smax):
                         for t in range(tmax):
                             v = x - smid + s
                             w = y - tmid + t
                             value += g[s, t] * f[v, w]
```

```
            h[x, y] = value

            #v1 = x - smid
            #v2 = v1 + smax
            #w1 = y - tmid
            #w2 = w1 + tmax
            #h[x, y] = np.sum(g * f[v1:v2, w1:w2])
    return h


@nb.jit(nopython=True, nogil=True, cache=False, parallel=True)
def convolve_numba(f, g):

    if g.shape[0] % 2 != 1 or g.shape[1] % 2 != 1:
        raise ValueError("Only odd dimensions on filter supported")

    return convolve_kernel(f, g)

# Stencil contains implicit loops
@nb.stencil(standard_indexing=("g",),neighborhood=((-4, 4),(-4, 4)))
def convolve_kernel1(f, g):

    smax = g.shape[0]
    tmax = g.shape[1]

    smid = smax // 2
    tmid = tmax // 2

    h = 0
    for s in range(smax):
        for t in range(tmax):
            h += g[s, t] * f[s - smid, t - tmid]

    return h

@nb.jit(nopython=True, nogil=True, cache=False, parallel=True)
def convolve_numba1(f, g):

    if g.shape[0] % 2 != 1 or g.shape[1] % 2 != 1:
        raise ValueError("Only odd dimensions on filter supported")

    return convolve_kernel1(f, g).astype(f.dtype)
```

In [32]:

```
# It will be compiled at first use
tmpa = np.arange(1, 5)
tmpb = np.arange(1, 5)
nb_function1(tmpa, tmpb)
nb_function2(tmpa, tmpb)
```

```
         N = 10
         tmpf = np.arange(N * N, dtype=np.int).reshape((N, N))
         _ = convolve_numba(tmpf, g)
         _ = convolve_numba1(tmpf, g)
         print("compilation OK")
```

```
compilation OK
```

```
In [33]: _ = check(nb_function1, (a, b), ref=ref1)
         _ = check(nb_function2, (a, b), ref=ref2)
         print()
         _ = check(convolve_numba, (small_f, g), ref=ref_small_convolve)
         _ = check(convolve_numba, (large_f, g), ref=ref_large_convolve)
         print()
         _ = check(convolve_numba1, (small_f, g), ref=ref_small_convolve)
         _ = check(convolve_numba1, (large_f, g), ref=ref_large_convolve)
```

```
Testing  nb_function1
797 ţs ś 8.59 ţs per loop (mean ś std. dev. of 7 runs, 1000 loops each)
Testing  nb_function2
14.1 ms ś 169 ţs per loop (mean ś std. dev. of 7 runs, 100 loops each)

Testing  convolve_numba
2.35 ms ś 23.6 ţs per loop (mean ś std. dev. of 7 runs, 100 loops each)
Testing  convolve_numba
251 ms ś 2.42 ms per loop (mean ś std. dev. of 7 runs, 1 loop each)

Testing  convolve_numba1
1.51 ms ś 10.9 ţs per loop (mean ś std. dev. of 7 runs, 1000 loops each)
Testing  convolve_numba1
101 ms ś 1.04 ms per loop (mean ś std. dev. of 7 runs, 10 loops each)
```

2.5 Cython is the most efficient way to optimize your code

But you have to: - type *every* variable - explicit all loops - parallelize manually - compile it separately (or use ipython magic) - dependencies - Windows : - Visual studio build tools - or Mingw - or Windows Subsystem for Linux - Linux : - gcc - or icc

```
In [34]: %load_ext Cython
```

```
In [35]: %%cython -a
         # cython: language_level=3
         # cython: initializedcheck=False
         # cython: binding=True
         # cython: nonecheck=False
         # distutils: extra_compile_args = -fopenmp -march=native -Ofast -fvect-cost-model=cheap
         # distutils: extra_link_args = -fopenmp
```

```
### cython: np_pythran=True
import numpy as np
cimport numpy as np
cimport cython
from libc.math cimport sin
from libc.math cimport asinh
from cython.parallel cimport parallel, prange

@cython.boundscheck(False) # turn off bounds-checking for entire function
@cython.wraparound(False)  # turn off negative index wrapping for entire function
def cfunction1(double[::1] a, double[::1] b):
    cdef long n = a.shape[0]

    cdef long[:] res = np.empty([n], dtype=long)

    cdef Py_ssize_t i
    for i in prange(n, nogil=True, num_threads=8):
        res[i] = a[i] * b[i] - 4.1 * a[i] > 2.5 * b[i]
    return np.asarray(res, dtype=bool)

@cython.boundscheck(False) # turn off bounds-checking for entire function
@cython.wraparound(False)  # turn off negative index wrapping for entire function
def cfunction2(double[::1] a, double[::1] b):
    cdef long n = a.shape[0]

    cdef double[:] res = np.empty([n], dtype=np.double)

    cdef Py_ssize_t i
    for i in prange(n, nogil=True, num_threads=8):
        res[i] = sin(a[i]) + asinh(a[i] / b[i])
    return np.asarray(res)


@cython.boundscheck(False) # turn off bounds-checking for entire function
@cython.wraparound(False)  # turn off negative index wrapping for entire function
def convolve_cython(long[:,::1]& f, long[:,::1]& g):
    cdef long vmax = f.shape[0]
    cdef long wmax = f.shape[1]
    cdef long smax = g.shape[0]
    cdef long tmax = g.shape[1]

    # f is an image and is indexed by (v, w)
    # g is a filter kernel and is indexed by (s, t),
    #   it needs odd dimensions
    # h is the output image and is indexed by (x, y),
    if smax % 2 != 1 or tmax % 2 != 1:
        raise ValueError("Only odd dimensions on filter supported")
```

```
             # smid and tmid are number of pixels between the center pixel
             # and the edge, ie for a 5x5 filter they will be 2.

             cdef long smid = smax // 2
             cdef long tmid = tmax // 2

             # Allocate result image.
             cdef long[:,::1] h = np.zeros([vmax, wmax], dtype=long)

             cdef long value
             cdef Py_ssize_t x, y, s, t, v, w

             # Do convolution
             for x in prange(smid, vmax - smid, nogil=True, num_threads=8):
                 for y in range(tmid, wmax - tmid):
                     # Calculate pixel value for h at (x,y). Sum one component
                     # for each pixel (s, t) of the filter g.

                     value = 0
                     for s in range(smax):
                         for t in range(tmax):
                             v = x - smid + s
                             w = y - tmid + t
                             value = value + g[s, t] * f[v, w]
                     h[x, y] = value

             return np.asarray(h)

Out[35]: <IPython.core.display.HTML object>

In [36]: _ = check(cfunction1, (a, b), ref=ref1)
         _ = check(cfunction2, (a, b), ref=ref2)
         print()
         _ = check(convolve_cython, (small_f, g), ref=ref_small_convolve)
         _ = check(convolve_cython, (large_f, g), ref=ref_large_convolve)

Testing  cfunction1
2.1 ms ś 296 ţs per loop (mean ś std. dev. of 7 runs, 100 loops each)
Testing  cfunction2
15 ms ś 126 ţs per loop (mean ś std. dev. of 7 runs, 100 loops each)

Testing  convolve_cython
655 ţs ś 6.15 ţs per loop (mean ś std. dev. of 7 runs, 1000 loops each)
Testing  convolve_cython
68.8 ms ś 608 ţs per loop (mean ś std. dev. of 7 runs, 10 loops each)


In [37]: csource = """
             #include "CModule.h"
```

```
    void convolve_c (const long f[],
                     const long g[],
                     long h[],
                     const size_t vmax,
                     const size_t wmax,
                     const size_t smax,
                     const size_t tmax)
    {

        const size_t smid = smax / 2;
        const size_t tmid = tmax / 2;

        for(size_t s = 0; s < vmax * wmax; ++s) {
            h[s] = 0;
        }

        // Do convolution
        #pragma omp parallel for default(shared) num_threads(8)
        for(size_t x = smid; x < vmax - smid; ++x) {
            for(size_t y = tmid; y < wmax - tmid; ++y) {
                // Calculate pixel value for h at (x,y).
                // Sum one component for each pixel (s, t) of the filter g.

                long value = 0;
                for(size_t s = 0; s < smax; ++s) {
                    for(size_t t = 0; t < tmax; ++t) {
                        size_t v = x - smid + s;
                        size_t w = y - tmid + t;
                        value = value + g[s*tmax + t] * f[v*wmax + w];
                    }
                }
                h[x*wmax + y] = value;
            }
        }
    }
    """

hsource = """
    #include <stddef.h>

    void convolve_c (const long f[],
                     const long g[],
                     long h[],
                     const size_t vmax,
                     const size_t wmax,
                     const size_t smax,
                     const size_t tmax);
    """

with open("CModule.c", 'w') as cfile:
    for line in csource:
        cfile.write(line)


with open("CModule.h", 'w') as hfile:
    for line in hsource:
        hfile.write(line)
```

In [38]:

```
%%cython  -a
# cython: language_level=3
# cython: initializedcheck=False
# cython: binding=True
# cython: nonecheck=False
# distutils: sources = CModule.c
# distutils: extra_compile_args = -fopenmp -march=native -Ofast -fvect-cost-model=cheap
# distutils: extra_link_args = -fopenmp
### cython: np_pythran=True
import numpy as np
cimport numpy as np
cimport cython

cdef extern from "CModule.h":
    long* convolve_c (const long f[],
                      const long g[],
                      long h[],
                      const size_t vmax,
                      const size_t wmax,
                      const size_t smax,
                      const size_t tmax) nogil

@cython.boundscheck(False) # turn off bounds-checking for entire function
@cython.wraparound(False)  # turn off negative index wrapping for entire function
def convolve_cython2(long[:,::1]& f, long[:,::1]& g):
    # f is an image and is indexed by (v, w)
    # g is a filter kernel and is indexed by (s, t),
    #   it needs odd dimensions
    # h is the output image and is indexed by (x, y),

    cdef long vmax = f.shape[0]
    cdef long wmax = f.shape[1]
    cdef long smax = g.shape[0]
    cdef long tmax = g.shape[1]

    if smax % 2 != 1 or tmax % 2 != 1:
        raise ValueError("Only odd dimensions on filter supported")
```

```
             cdef long[:,::1] h = np.empty([vmax, wmax], dtype=long)

             # Do convolution
             with nogil:
                  convolve_c(&f[0,0],
                             &g[0,0],
                             &h[0,0], vmax, wmax, smax, tmax)


             return np.asarray(h)

Out[38]: <IPython.core.display.HTML object>

In [39]: _ = check(convolve_cython2, (small_f, g), ref=ref_small_convolve)
         _ = check(convolve_cython2, (large_f, g), ref=ref_large_convolve)

Testing  convolve_cython2
566 ţs ś 19.1 ţs per loop (mean ś std. dev. of 7 runs, 1000 loops each)
Testing  convolve_cython2
56.4 ms ś 509 ţs per loop (mean ś std. dev. of 7 runs, 10 loops each)
```

2.6 Fortran through f2py is also very efficient

But you have to - rewrite your code - compile it separately

```
In [40]: fsource = """
             module fortranmodule
             use OMP_LIB
             implicit none
             contains

             subroutine function1(a, b, c, n)
             implicit none
             integer(kind=8), intent(in) :: n
             double precision,intent(in) :: a(n)
             double precision,intent(in) :: b(n)

             logical,intent(out)         :: c(n)

             !$OMP PARALLEL WORKSHARE NUM_THREADS(8)
             c = a * b - 4.1 * a > 2.5 * b
             !$OMP END PARALLEL WORKSHARE

             end subroutine function1

             subroutine function2(a, b, c, n)
             implicit none
```

```
integer(kind=8), intent(in)  :: n
double precision,intent(in)  :: a(n)
double precision,intent(in)  :: b(n)

double precision,intent(out) :: c(n)

!$OMP PARALLEL WORKSHARE NUM_THREADS(8)
c = sin(a) + asinh(a / b)
!$OMP END PARALLEL WORKSHARE

end subroutine function2


subroutine convolve_fortran(f, g, vmax, wmax, smax, tmax, h, err)
implicit none
integer(kind=8),intent(in)  :: vmax,wmax,smax,tmax
integer(kind=8),intent(in)  :: f(vmax, wmax), g(smax, tmax)

integer(kind=8),intent(out) :: h(vmax, wmax)
integer(kind=8),intent(out) :: err

integer(kind=8) :: smid,tmid
!integer(kind=8) :: value
integer(kind=8) :: x, y
!integer(kind=8) :: v, w, s, t
integer(kind=8) :: v1,v2,w1,w2

! f is an image and is indexed by (v, w)
! g is a filter kernel and is indexed by (s, t),
!   it needs odd dimensions
! h is the output image and is indexed by (v, w),

err = 0
if (modulo(smax, 2) /= 1 .or. modulo(tmax, 2) /= 1) then
    err = 1
    return
endif

! smid and tmid are number of pixels between the center pixel
! and the edge, ie for a 5x5 filter they will be 2.
smid = smax / 2
tmid = tmax / 2

h = 0
! Do convolution
! warning : memory layout is different in fortran
! warning : array start at 1 in fortran
```

```
    !$OMP PARALLEL DO DEFAULT(SHARED) COLLAPSE(1) &
    !$OMP PRIVATE(v1,v2,w1,w2) NUM_THREADS(8)
    do y = tmid + 1,wmax - tmid
        do x = smid + 1,vmax - smid
            ! Calculate pixel value for h at (x,y). Sum one component
            ! for each pixel (s, t) of the filter g.

            !value = 0
            !do t = 1, tmax
            !    do s = 1, smax
            !        v = x - smid + s -1
            !        w = y - tmid + t -1
            !
            !        value = value + g(s, t) * f(v, w)
            !    enddo
            !enddo

            v1 = x - smid
            v2 = v1 + smax
            w1 = y - tmid
            w2 = w1 + tmax
            h(x, y) = sum(g(1:smax,1:tmax) * f(v1:v2,w1:w2))
        enddo
    enddo
    !$OMP END PARALLEL DO
    return
    end subroutine convolve_fortran
    end module fortranmodule
    """

with open("fortranModule.f90", 'w') as fortranfile:
    for line in fsource:
        fortranfile.write(line)

import subprocess
try:
    data = subprocess.check_output(["f2py",
#    data = subprocess.check_output(["/home/pythonstudent/.local/bin/f2py",
                                    "-c", "fortranModule.f90",
                                    "-m", "myfortranmodule",
                                    "--opt='-Ofast -march=native -fopenmp -fvect-cost-m
                                    #"--opt='-Ofast -xHost -qopenmp '", "--noarch", "-l
                                    #"--debug-capi", "--debug",
                                    "-DF2PY_REPORT_ON_ARRAY_COPY=1"
                                   ],stderr=subprocess.STDOUT).decode('utf-8')
except subprocess.CalledProcessError as e:
    print(e.output.decode('utf-8'))
else:
```

```
            #print(data)
            print("compilation OK")
```

```
compilation OK
```

```
In [41]: import os, sys
         from myfortranmodule import fortranmodule

         _ffunction1 = fortranmodule.function1
         ffunction2 = fortranmodule.function2
         ffunction2.__name__ = "ffunction2"
         convolve_fortran = fortranmodule.convolve_fortran

         def ffunction1(a ,b):
             return _ffunction1(a, b).astype(bool)

         def fortran_convolve(f, g):
             h, err = convolve_fortran(f, g)
             if err:
                 print(err)
                 raise ValueError("FORTRAN ERROR ! (Probably : Only odd dimensions on filter sup
             return h

         def fortran_setup(f, g):
             # memory ordering for fortran
             ft = np.asfortranarray(f.copy())
             gt = np.asfortranarray(g.copy())
             return ft, gt
```

```
In [42]: ### The produced binary is simple enough to be decompiled
         import subprocess
         data = subprocess.check_output(["gdb", "-batch",
                                         "-ex", "file myfortranmodule.cpython-35m-x86_64-linux-g
                                         "-ex", "disassemble /s __fortranmodule_MOD_function1"])
                                         #"-ex", "disassemble /m fortranmodule_mp_function1_"]).
         print(data)
```

```
Dump of assembler code for function __fortranmodule_MOD_function1:
fortranModule.f90:
7             subroutine function1(a, b, c, n)
   0x0000000000005cd0 <+0>:        sub    $0x38,%rsp
   0x0000000000005cd4 <+4>:        mov    (%rcx),%rax
   0x0000000000005cd7 <+7>:        xor    %ecx,%ecx

8             implicit none
9             integer(kind=8), intent(in) :: n
10             double precision,intent(in) :: a(n)
11             double precision,intent(in) :: b(n)
```

```
12
13              logical,intent(out)         :: c(n)
14
15              !$OMP PARALLEL WORKSHARE NUM_THREADS(8)
   0x0000000000005cd9 <+9>:        mov    %rdi,(%rsp)
   0x0000000000005cdd <+13>:        lea    -0x504(%rip),%rdi        # 0x57e0 <__fortranmodule_MO
   0x0000000000005ce4 <+20>:        mov    %rsi,0x8(%rsp)
   0x0000000000005ce9 <+25>:        mov    %rdx,0x10(%rsp)
   0x0000000000005cee <+30>:        mov    %rsp,%rsi
   0x0000000000005cf1 <+33>:        mov    $0x8,%edx
   0x0000000000005cf6 <+38>:        mov    %rax,0x28(%rsp)
   0x0000000000005cfb <+43>:        mov    %rax,0x20(%rsp)
   0x0000000000005d00 <+48>:        mov    %rax,0x18(%rsp)
   0x0000000000005d05 <+53>:        callq  0x1f50 <GOMP_parallel@plt>

16              c = a * b - 4.1 * a > 2.5 * b
17              !$OMP END PARALLEL WORKSHARE
18
19              end subroutine function1
   0x0000000000005d0a <+58>:        add    $0x38,%rsp
   0x0000000000005d0e <+62>:        retq
End of assembler dump.
```

```
In [43]: _ = check(ffunction1, (a, b), ref=ref1)
         _ = check(ffunction2, (a, b), ref=ref2)
         print()
         _ = check(fortran_convolve, (small_f, g), ref=ref_small_convolve, setup=fortran_setup)
         _ = check(fortran_convolve, (large_f, g), ref=ref_large_convolve, setup=fortran_setup)
```

```
Testing  ffunction1
1.23 ms ś 12 ţs per loop (mean ś std. dev. of 7 runs, 1000 loops each)
Testing  ffunction2
11.5 ms ś 115 ţs per loop (mean ś std. dev. of 7 runs, 100 loops each)

Testing  fortran_convolve
566 ţs ś 4.39 ţs per loop (mean ś std. dev. of 7 runs, 1000 loops each)
Testing  fortran_convolve
59.7 ms ś 1.15 ms per loop (mean ś std. dev. of 7 runs, 10 loops each)
```

- parallelism
- cuda

```
In [44]: """
         CUDA DOESN'T WORK ON THE VIRTUAL MACHINE
         YOU ARE WELCOME TO TRY THIS ON YOU OWN COMPUTER
         """
         from string import Template

         cuda_src_template = Template("""
         // Cuda splitting
         #define MTB ${max_threads_per_block}
         #define MBP ${max_blocks_per_grid}

         // Array size
         #define fx ${fx}
         #define fy ${fy}
         #define gx ${gx}
         #define gy ${gy}

         // Macro for converting subscripts to linear index:
         #define f_INDEX(i, j) (i) * (fy) + (j)

         // Macro for converting subscripts to linear index:
         #define g_INDEX(i, j) (i) * (gy) + (j)

         __global__ void convolve_cuda(long *f, long *g, long *h) {

             unsigned int idx = blockIdx.y * MTB * MBP + blockIdx.x * MTB + threadIdx.x;

             // Convert the linear index to subscripts:
             unsigned int i = idx / fy;
             unsigned int j = idx % fy;

             long smax = gx;
             long tmax = gy;

             long smid = smax / 2;
             long tmid = tmax / 2;

             if (smid <= i && i < fx - smid) {
             if (tmid <= j && j < fy - tmid) {

                 h[f_INDEX(i, j)] = 0.;

                 for (long s = 0; s < smax; s++)
                     for (long t = 0; t < tmax; t++)
                         h[f_INDEX(i, j)] += g[g_INDEX(s, t)] * f[f_INDEX(i + s - smid, j + t -

             }
             }
         }
         """)

In [45]: """
```

```
CUDA DOESN'T WORK ON THE VIRTUAL MACHINE
YOU ARE WELCOME TO TRY THIS ON YOU OWN COMPUTER
"""
import skcuda.misc as misc
import pycuda.autoinit
device = pycuda.autoinit.device
max_threads_per_block, _, max_grid_dim = misc.get_dev_attrs(device)
max_blocks_per_grid = max(max_grid_dim)
```

```
    ---------------------------------------------------------------------------

    Error                                     Traceback (most recent call last)

    /Data/WORK/Formations/Python/formation_python/04.advanced.ipynb in <module>()
      4 """
      5 import skcuda.misc as misc
----> 6 import pycuda.autoinit
      7 device = pycuda.autoinit.device
      8 max_threads_per_block, _, max_grid_dim = misc.get_dev_attrs(device)


    /Data/WORK/Formations/Python/formation_python/venv/lib/python3.5/site-packages/pycuda/au
      3
      4 # Initialize CUDA
----> 5 cuda.init()
      6
      7 from pycuda.tools import make_default_context


    Error: cuInit failed: unknown error
```

```
In [46]: """
         CUDA DOESN'T WORK ON THE VIRTUAL MACHINE
         YOU ARE WELCOME TO TRY THIS ON YOU OWN COMPUTER
         """
         from functools import partial
         from pycuda.compiler import SourceModule

         cuda_src = cuda_src_template.substitute(max_threads_per_block=max_threads_per_block,
                                                 max_blocks_per_grid=max_blocks_per_grid,
                                                 fx=large_f.shape[0], fy=large_f.shape[1],
                                                 gx=g.shape[0], gy=g.shape[1]
                                                )
         cuda_module = SourceModule(cuda_src, options= ["-O3", "-use_fast_math", "-default-strea
         print("Compilation OK")
```

```
    __convolve_cuda = cuda_module.get_function('convolve_cuda')

    block_dim, grid_dim = misc.select_block_grid_sizes(device, f.shape)
    _convolve_cuda = partial(__convolve_cuda,
                             block=block_dim,
                             grid=grid_dim)
```

```
---------------------------------------------------------------------------

NameError                                 Traceback (most recent call last)

/Data/WORK/Formations/Python/formation_python/04.advanced.ipynb in <module>()
      6 from pycuda.compiler import SourceModule
      7
----> 8 cuda_src = cuda_src_template.substitute(max_threads_per_block=max_threads_per_block,
      9                                          max_blocks_per_grid=max_blocks_per_grid,
     10                                          fx=large_f.shape[0], fy=large_f.shape[1],


NameError: name 'max_threads_per_block' is not defined
```

```
In [47]: """
         CUDA DOESN'T WORK ON THE VIRTUAL MACHINE
         YOU ARE WELCOME TO TRY THIS ON YOU OWN COMPUTER
         """
         import pycuda.gpuarray as gpuarray

         def convolve_cuda(f, g):
             f_gpu, g_gpu = cuda_setup(f, g)
             h_gpu = convolve_cuda2(f_gpu, g_gpu)
             return cuda_finish(h_gpu)

         def convolve_cuda2(f_gpu, g_gpu):
             h_gpu = gpuarray.zeros_like(f_gpu)
             _convolve_cuda(f_gpu, g_gpu, h_gpu)
             return h_gpu

         def cuda_setup(f, g):
             f_gpu = gpuarray.to_gpu(f)
             g_gpu = gpuarray.to_gpu(g)
             return f_gpu, g_gpu

         def cuda_finish(h_gpu):
             return h_gpu.get()
```

```
In [48]: """
         CUDA DOESN'T WORK ON THE VIRTUAL MACHINE
```

```
YOU ARE WELCOME TO TRY THIS ON YOU OWN COMPUTER
"""
print("Cuda")
check(convolve_cuda, (f, g), ref=ref)
print("Cuda without comm")
check(convolve_cuda2, (f, g), ref=ref, setup=cuda_setup, finish=cuda_finish)
print("Finished")
```

```
Cuda
Testing  convolve_cuda
```

```
    ---------------------------------------------------------------------------

    LogicError                                Traceback (most recent call last)

    /Data/WORK/Formations/Python/formation_python/04.advanced.ipynb in <module>()
      4 """
      5 print("Cuda")
----> 6 check(convolve_cuda, (f, g), ref=ref)
      7 print("Cuda without comm")
      8 check(convolve_cuda2, (f, g), ref=ref, setup=cuda_setup, finish=cuda_finish)


    /Data/WORK/Formations/Python/formation_python/04.advanced.ipynb in check(fun, data, ref,
     12
     13     if ref is None:
---> 14         res = finish(fun(f, g))
     15     else:
     16         res = finish(fun(f, g))


    /Data/WORK/Formations/Python/formation_python/04.advanced.ipynb in convolve_cuda(f, g)
      6
      7 def convolve_cuda(f, g):
----> 8     f_gpu, g_gpu = cuda_setup(f, g)
      9     h_gpu = convolve_cuda2(f_gpu, g_gpu)
     10     return cuda_finish(h_gpu)


    /Data/WORK/Formations/Python/formation_python/04.advanced.ipynb in cuda_setup(f, g)
     16
     17 def cuda_setup(f, g):
---> 18     f_gpu = gpuarray.to_gpu(f)
     19     g_gpu = gpuarray.to_gpu(g)
     20     return f_gpu, g_gpu
    /Data/WORK/Formations/Python/formation_python/venv/lib/python3.5/site-packages/pycuda/gp
    990 def to_gpu(ary, allocator=drv.mem_alloc):
    991     """converts a numpy array to a GPUArray"""
--> 992     result = GPUArray(ary.shape, ary.dtype, allocator, strides=_compact_strides(ary)
    993     result.set(ary)
    994     return result


    /Data/WORK/Formations/Python/formation_python/venv/lib/python3.5/site-packages/pycuda/gp
    208         if gpudata is None:
    209             if self.size:
--> 210                 self.gpudata = self.allocator(self.size * self.dtype.itemsize)
    211             else:
    212                 self.gpudata = None


    LogicError: cuMemAlloc failed: initialization error
```

- asyncio
  - not a real parallelism
  - effective for io-bound tasks (web)
  - not very interesting here
- multithreading
  - more parallelism (GIL)
  - shared memory
  - two main implementation
    - threading (stdlib) which is flexible
    - joblib which is relatively easy to use
- multiprocessing
  - real parallelism
  - limited to one computer
  - two main implementation
    - multiprocessing (stdlib) which is flexible
    - joblib which is relatively easy to use
- mpi (mpi4py)
  - real parallelism
  - unlimited
  - relatively complex to use (same as in C, fortran, ...)

```
In [49]: import time
         import numpy as np
         def heavy_fonction(i):
             t = np.random.rand() / 10
             time.sleep(t)
             return i, t
```

```
In [50]: from joblib import Parallel, delayed

         if __name__ == "__main__":

             tic = time.time()
             res = Parallel(n_jobs=-1, backend='threading')(delayed(heavy_fonction)(i) \
                                         for i in range(2000))
             tac = time.time()
             index, times = np.asarray(res).T
             print(tac - tic)
             print(times.sum())
```

```
12.737246036529541
100.36369606025077
```

```
In [51]: from threading import Thread, RLock

         N = 2000
         N_t = 10
         current = 0
         nprocs = 8
         output_list = np.empty(N)

         lock = RLock()

         class ThreadJob(Thread):
             def run(self):
                 """This code will be executed by each thread"""
                 global current

                 while current < N:

                     with lock:
                         position = current
                         current += N_t

                     fin = min(position + N_t + 1, N)

                     for i in range(position, fin):
                         j, t = heavy_fonction(i)
                         output_list[j] = t
         if __name__ == "__main__":

             # Threads creation
             threads = [ThreadJob() for i in range(nprocs)]

             tic = time.time()
             # Threads starts
             for thread in threads:
                 thread.start()

             # Waiting that all thread have finish
             for thread in threads:
                 thread.join()
             tac = time.time()


             print(tac - tic)
             print(output_list.sum())
```

```
14.059715747833252
99.97539302441548
```

```
In [52]: import multiprocessing as mp
         from queue import Empty

         def process_job(q,r):
             """This code will be executed by each process"""
             while True:
                 try:
                     i = q.get(block=False)
                     r.put(heavy_fonction(i))
                 except Empty:
                     if q.empty():
                         if q.qsize() == 0:
                             break

         if __name__ == "__main__":

             # Define an output queue
             r = mp.Queue()

             # Define an input queue
             q = mp.Queue()

             for i in range(2000):
                 q.put(i)
```

```
nprocs = 8
# Setup a list of processes that we want to run
processes = [mp.Process(target=process_job, args=(q, r)) for i in range(nprocs)]

tic = time.time()

# Run processes
for p in processes:
    p.start()

# Get process results from the output queue
results = np.empty(2000)
for i in range(2000):
    j, t = r.get()
    results[j] = t

tac = time.time()

# Exit the completed processes
for p in processes:
    p.join()

print(tac - tic)
print(results.sum())
```

```
12.583194017410278
99.33507414916964
```

2.6.1 Exercise

The following code read an image in pgm format (ascii) and store it in a 2D list.
For each pixel of the image a kernel get all neighbors (9 counting the pixel itself) and apply a computation.
Analyse the performance of the code, identify bottleneck and try to optimize it.

```
In [53]: %matplotlib notebook

         import math
         import matplotlib.pyplot as plt

         def get_description(filename):
             """
             Read the header part of the file
             """
             f = open(filename, 'r')
             nline = 0
             description = {}
```

```
    while nline < 3:
        line = f.readline()
        if line[0] == '#':
            continue
        nline += 1
        if nline == 1:
            description['format'] = line.strip()
        elif nline == 2:
            description['dimension'] = int(line.split()[1]), int(line.split()[0])
        elif nline ==3:
            description['deep'] = int(line.strip())
    f.close()
    return description

def get_value(filename, coord):
    """
    Get value at coord in an image in the PGM format

    The main problem here is that the file have a limited width, and the values are wra
    Thus, the value at coord 12,32 might be in the 24,6 in the file
    """
    description = get_description(filename)
    xdim, ydim = description['dimension']
    i = coord[0]
    j = coord[1]
    f = open(filename, 'r', encoding='utf-8')
    nline = 0
    while nline < 3:
        line = f.readline()
        if line[0] == '#':
            continue
        nline += 1
    #here we are at coordinate (0,0)
    icur, jcur = 0,0
    test = True
    while(test):
        values = f.readline().split()
        nvalues = len(values)
        if (icur == i):
            if (jcur + nvalues > j):
                jvalues = j - jcur
                value = values[jvalues]
                test=False
            else:
                jcur += nvalues
        else:
            jcur += nvalues
        if (jcur >= ydim):
            icur += jcur // ydim
            jcur = jcur % ydim
    f.close()
    return int(value)


def read_file(filename):
    """
    Read an image in the PGM format
    """
    description=get_description(filename)
    values = []
    for i in range(description['dimension'][0]):
        values.append([])
        for j in range(description['dimension'][1]):
            values[i].append(get_value(filename, (i, j)))

    return values

def get_neighbors(tab, i, j):
    neigh = []
    for jrange in [-1, 0, 1]:
        for irange in [-1, 0, 1]:
            neigh.append(tab[i + irange][j + jrange])
    return neigh

def compute_wtf(neigh):
    value = 1.
    for i in range(len(neigh)):
        value *= math.exp(neigh[i]) ** (1 / len(neigh))
    value = math.log(value)

    return float(value)

def kernel(tab):
    """
    Apply compute_wtf on each pixel except boundary
    """
    xdim = len(tab)
    ydim = len(tab[0])
    result = []
    #1st line
    result.append([0])
    for jrange in range(1, ydim):
        result[0].append(0)
    for irange in range(1, xdim - 1):
        #1st column
        result.append([0])
        for jrange in range(1, ydim - 1):
            neigh = get_neighbors(tab, irange, jrange)
            result[irange].append(compute_wtf(neigh))
        #last colum
        result[irange].append(0)
    #last line
    result.append([])
    for jrange in range(ydim):
        result[xdim - 1].append(0)
    return result

def job(data):
    """
    Apply kernel of each image
    """
    results = []
    for image in data:
        results.append(kernel(image))
    return results

def init(files):
    """
    Read all files
    """
    data = []
    for file in files:
        data.append(read_file(file))

    return data

def plot(data):
    nimages = len(data)

    if nimages > 1:
        fig, axes = plt.subplots(nimages, 1)
        for image, ax in zip(data, axes):
            ax.imshow(image)
    else:
        plt.figure()
        plt.imshow(data[0])

    plt.show()


files=["data/test.pgm"]

if __name__ == "__main__":
        print("Reading Files")
        data = init(files)

        print("Computing")
        result = job(data)

        plot(data)
        plot(result)
```

```
Reading Files
Computing


<IPython.core.display.Javascript object>


<IPython.core.display.HTML object>


<IPython.core.display.Javascript object>


<IPython.core.display.HTML object>
```

Try to optimize this code.
You can apply your code on the following images : - data/test.pgm - data/test32.pgm - data/brain_604.ascii.pgm - data/apollonian_gasket.ascii.pgm - data/dla.ascii.pgm

For reference, the timing on my computer are :
For data/test.pgm

On my computer :

```
Reading Files
6.67 ms ś 262 ţs per loop (mean ś std. dev. of 7 runs, 100 loops each)
Computing
503 ţs ś 5.41 ţs per loop (mean ś std. dev. of 7 runs, 1000 loops each)
```

My solution

```
Reading Files
83.9 ţs ś 3.9 ţs per loop (mean ś std. dev. of 7 runs, 10000 loops each)
Computing
255 ţs ś 19.4 ţs per loop (mean ś std. dev. of 7 runs, 1000 loops each)
```

And, the bigger the image, the bigger the gain !

2.6.2 Hints

Part 1 - Open input file only once - Avoid appending data - Use numpy array for data storage - What is really doing the compute_wtf function ?

Part 2 - compile the compute part

Part 3 - parallelize the work on each image

2.6.3 Solution part 1

```
import numpy as np
from copy import deepcopy

def get_description(file):
    """
    Read the header part of the file

    And leave the file at the end of header (start of values)
    """
    # return to begining
    file.seek(0)
    nline = 0
    description = {}
    while nline < 3:
        line = file.readline()
        if line[0] == '#':
            continue
        nline += 1
        if nline == 1:
            description['format'] = line.strip()
        elif nline == 2:
            description['dimension'] = int(line.split()[1]), int(line.split()[0])
        elif nline == 3:
            description['deep'] = int(line.strip())
    return description

def read_values(file, description):
    """
    Read all the values directly
    """
    # pre-allocate the array
    nx, ny = description['dimension']
    values = np.empty((nx * ny))
    i = 0
    for line in file:
        if line[0] == '#':
            continue
        vals = line.split()
        nvals = len(vals)
        values[i:i + nvals] = [int(v) for v in vals]
        i += nvals
    return values.reshape((nx, ny))

def read_file(filename):
    """
    Read an image in the PGM format
    """
    # open the file once
    with open(filename, 'r', encoding="utf-8") as file:

        # read the header part
        description = get_description(file)

        # read the values
        values = read_values(file, description)
    return values


def init(files):
    """
    Read all files
    """
    data = []
    for file in files:
        data.append(read_file(file))

    return data

#prepare result array
result = deepcopy(data)
```

2.6.4 Solution part 2

In the python part :

```
def kernel(i):
    """
    Apply compute_wtf on each pixel except boundary
    """
    global data, result, t_omp
    result[i] = ckernel(data[i], nt_omp)
```

In a new cell : `%load_ext Cython`

In a new cell :

```
%%cython --compile-args=-fopenmp --link-args=-fopenmp
## --compile-args=-DCYTHON_TRACE_NOGIL=1 --compile-args=-DCYTHON_TRACE=1
# cython: language_level=3
# cython: boundscheck=False
# cython: wraparound=False
# cython: initializedcheck=False
# cython: binding=True
import numpy as np
```

```
cimport numpy as np
cimport cython
from cython.parallel cimport parallel, prange

def ckernel(double[:,::1] data,long nt):
    cdef long n = data.shape[0]
    cdef long m = data.shape[1]

    cdef double[:,::1] res = np.zeros([n, m], dtype=np.double)
    cdef double value

    cdef long i, j, s, t
    with nogil, parallel(num_threads=nt):
        for i in prange(1, n - 1):
            for j in range(1, m - 1):
                value = 0
                for s in range(-1, 2):
                    for t in range(-1, 2):
                        value += data[i + s, j + t]
                res[i, j] += value / 9
    return res
```

2.6.5 Solution part 3

```
from threading import Thread, RLock
import os

nprocs = os.cpu_count()
nt_omp = nprocs // 2
nt_job =nprocs - nt_omp

result = []
data = []

current = 0

verrou = RLock()

class ThreadJob(Thread):
    def run(self):
        global current,verrou
        """Code à exécuter pendant l'exécution du thread."""
        while current < len(data):

            with verrou:
                position = current
                current += 1
            kernel(position)

def job(data):
    """
    Apply kernel on each image
    """
    global current

    current = 0
    # Création des threads
    threads = [ThreadJob() for i in range(nt_job)]

    # Lancement des threads
    for thread in threads:
        thread.start()

    # Attend que les threads se terminent
    for thread in threads:
        thread.join()


#sort data bigger first for better equilibrium
data = sorted(data, key=np.size, reverse=True)
```

2.6.6 Full Solution

Cell 1 : `%load_ext Cython`

Cell 2 :

```
%%cython --compile-args=-fopenmp --link-args=-fopenmp
## --compile-args=-DCYTHON_TRACE_NOGIL=1 --compile-args=-DCYTHON_TRACE=1
# cython: language_level=3
# cython: boundscheck=False
# cython: wraparound=False
# cython: initializedcheck=False
# cython: binding=True
import numpy as np
cimport numpy as np
cimport cython
from cython.parallel cimport parallel, prange

def ckernel(double[:,::1] data, long nt):
    cdef long n = data.shape[0]
    cdef long m = data.shape[1]

    cdef double[:,::1] res = np.zeros([n, m], dtype=np.double)
    cdef double value
    cdef long i, j, s, t
    with nogil, parallel(num_threads=nt):
        for i in prange(1, n - 1):
            for j in range(1, m - 1):
                value = 0
                for s in range(-1, 2):
                    for t in range(-1, 2):
                        value += data[i + s, j + t]
                res[i, j] += value / 9
    return res
```

Cell 3 :

```
%matplotlib notebook

import numpy as np
import matplotlib.pyplot as plt
from threading import Thread, RLock
from copy import deepcopy
import os

nprocs = os.cpu_count()
nt_omp = nprocs // 2
nt_job =nprocs - nt_omp

result = []
data = []

current = 0

verrou = RLock()

class ThreadJob(Thread):
    def run(self):
        global current,verrou
        """Code à exécuter pendant l'exécution du thread."""
        while current < len(data):

            with verrou:
                position = current
                current += 1

            kernel(position)

def get_description(file):
    """
    Read the header part of the file
    And leave the file at the end of header (start of values)
    """
    # return to begining
    file.seek(0)
    nline = 0
    description = {}
    while nline < 3:
        line = file.readline()
        if line[0] == '#':
            continue
        nline += 1
        if nline == 1:
            description['format']=line.strip()
        elif nline == 2:
            description['dimension']=int(line.split()[1]), int(line.split()[0])
        elif nline == 3:
            description['deep']=int(line.strip())
    return description

def read_values(file, description):
    """
    Read all the values directly
    """
    # pre-allocate the array
    nx, ny = description['dimension']
    values = np.empty((nx * ny))
    i = 0
    for line in file:
        if line[0] == '#':
            continue
        vals = line.split()
        nvals = len(vals)
        values[i:i + nvals] = [int(v) for v in vals]
        i += nvals
    return values.reshape((nx, ny))

def read_file(filename):
    """
    Read an image in the PGM format
    """
    # open the file once
    with open(filename, 'r', encoding="utf-8") as file:

        # read the header part
        description = get_description(file)

        # read the values
        values = read_values(file, description)
    return values

def kernel(i):
    """
    Apply compute_wtf on each pixel except boundary
    """
    global data, result, t_omp
    result[i] = ckernel(data[i], nt_omp)

def job(data):
    """
    Apply kernel of each image
    """
    global current

    current = 0
    # Création des threads
    threads = [ThreadJob() for i in range(nt_job)]

    # Lancement des threads
    for thread in threads:
        thread.start()

    # Attend que les threads se terminent
    for thread in threads:
        thread.join()


def init(files):
    """
    Read all files
    """
    data = []
    for file in files:
        data.append(read_file(file))

    return data

def plot(data):
    nimages = len(data)
    if nimages > 1:
        fig, axes = plt.subplots(nimages, 1)
        for image, ax in zip(data, axes):
            ax.imshow(image)
    else:
        plt.figure()
        plt.imshow(data[0])
    plt.show()


files = ["data/test.pgm",
          "data/test32.pgm",
          "data/brain_604.ascii.pgm",
          "data/apollonian_gasket.ascii.pgm",
          "data/dla.ascii.pgm",
        ]

if __name__ == "__main__":
    data = init(files)

    #sort data bigger first for better equilibrium
    data = sorted(data, key=np.size, reverse=True)

    #prepare result array
    result = deepcopy(data)

    #plot(data)
    plot(result)
```