Avanced Python

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

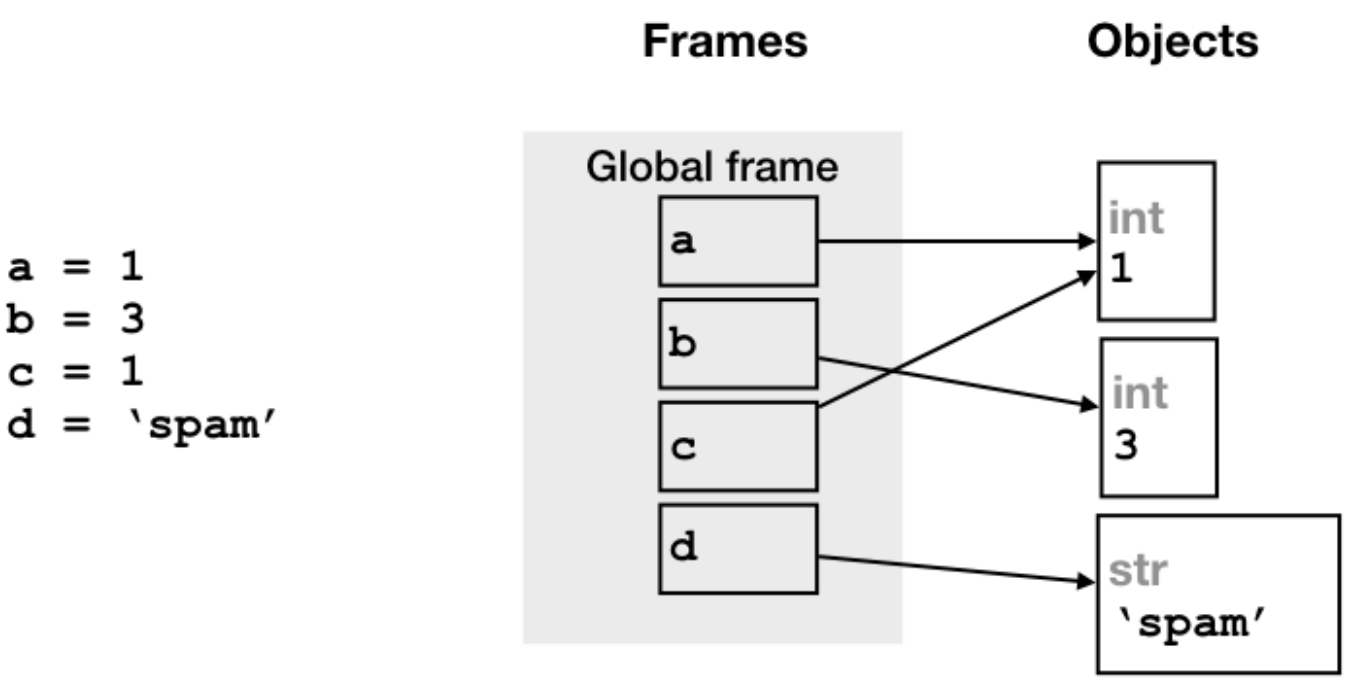
In [1]:

```
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
```



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

In [4]:

```
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())
```

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

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

In [5]:

```
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)

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

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

In [6]:

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

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

```
list Nok  ['einstein', 'bert', 'curie']

---------------------------------------------------------------------
--------
RuntimeError                              Traceback (most recent ca
ll last)
<ipython-input-6-a8611e4458eb> in <module>()
      6
      7 print("list Nok ", wrong(["einstein", "albert", "bert", "cu
rie"]))
----> 8 print("set  Nok ", wrong({"einstein", "albert", "bert", "cu
rie"}))
      9 print("END")

<ipython-input-6-a8611e4458eb> in wrong(my_set)
      1 def wrong(my_set):
----> 2     for value in my_set:
      3         if 'bert' in value:
      4             my_set.remove(value)
      5     return(my_set)

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 [7]:

```
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 [8]:

```
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 ca
ll last)
<ipython-input-8-4fd9146a2d0c> 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 [9]:

```
from functools import wraps
from time import time

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

    def after_f():
        new_f.tout = time()
        new_f.tcum += new_f.tout - new_f.tin
        print("Exiting {}".format(f.__name__))
        print("This was the call n° {}".format(new_f.NbAppels))
        print("It took {} s".format(new_f.tout - new_f.tin))
        print("in average {} s".format(new_f.tcum / new_f.NbAppels))

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

    new_f.NbAppels = 0
    new_f.tcum = 0.

    return new_f
```

In [10]:

```
import numpy as np
from time import sleep
import numpy as np
@PrintAppel
def a_function(x):
    np.random.rand()
    sleep(np.random.rand())
    return 2 * x
```

In [11]:

```
res = a_function(2)
print(res)

print(a_function.tcum / a_function.NbAppels)
```

```
Entering a_function
Exiting a_function
This was the call n° 1
It took 0.08617353439331055 s
in average 0.08617353439331055 s
4
0.08617353439331055
```

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

In [12]:

```
class egg(object):                                  # All objects derived from the s
ame object "object"
    """ Full exemple of a class in python """
    total_number = 0                                # shared attribut between all in
stances **DANGER** !

    def __init__(self, number=1):                   # constructor
        """ constructor from number """
        self.number = number                        # Good way of defining attribute
s
        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 print
able string
        """ egg to str convertor """
        return "On a total of {} eggs, I own {}".format(egg.total_number, self.n
umber)

    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 (rar
e)
        """ Return the total number of eggs for all recipes """
        return egg.total_number
```

In [13]:

```
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()))
```

```
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
```

In [14]:

```
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)
```

```
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

```
subprocess.check_call(["cmd", "arg1", "arg2"])
# or in jupyter
!cmd arg1 arg2
```

  - otherwise (remember to decode)

```
data = subprocess.check_output(["cmd", "arg1", "arg2"]).deco
de('utf-8')
```

In [15]:

```
!python3 script.py Marc

import subprocess
import sys
data = subprocess.check_output([sys.executable, "script.py", "Marc"]).decode('ut
f-8')
print(data)
```

```
OH HI MARC
OH HI MARC
```

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 [16]:

```
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 [17]:

```
#ONLY for ipython
import ipytest.magics
import pytest
__file__ = '04.advanced.ipynb'
```

In [18]:

```
%%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
```

```
...


   [100%]
```

- 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 [19]:

```
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] [%(leve
lname)-5.5s]  %(message)s")

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

In [20]:

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

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:2: UserWarning: A warning only once
  "cells": [
2018-03-26 16:13:33,461 [MainThread  ] [WARNI]  There is only 2 rec
ord !
2018-03-26 16:13:33,462 [MainThread  ] [ERROR]  Something happend,
impossible to save the records
2018-03-26 16:13:33,463 [MainThread  ] [CRITI]  Database corrupted
!
```

Performance

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

In [21]:

```
%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)]
```

```
465 µs ± 6.76 µs per loop (mean ± std. dev. of 7 runs, 1000 loops e
ach)
504 µs ± 10.3 µs per loop (mean ± std. dev. of 7 runs, 1000 loops e
ach)
489 µs ± 6.96 µs per loop (mean ± std. dev. of 7 runs, 1000 loops e
ach)
413 µs ± 4.76 µs per loop (mean ± std. dev. of 7 runs, 1000 loops e
ach)
```

In [22]:

```
%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)]
```

```
1.6 ms ± 34.9 µs per loop (mean ± std. dev. of 7 runs, 1000 loops e
ach)
1.68 ms ± 19.9 µs per loop (mean ± std. dev. of 7 runs, 1000 loops
each)
1.59 ms ± 10.8 µs per loop (mean ± std. dev. of 7 runs, 1000 loops
each)
2.24 ms ± 45.1 µs per loop (mean ± std. dev. of 7 runs, 100 loops e
ach)
```

- cProfile (for real code)

In [23]:

```
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")

# or in jupyter

%prun function1()
```

```
         7 function calls in 4.959 seconds

   Ordered by: internal time

   ncalls  tottime  percall  cumtime  percall filename:lineno(funct
ion)
        1    4.642    4.642    4.642    4.642 <ipython-input-23-7e4
471985ba7>:5(function2)
        1    0.239    0.239    0.239    0.239 {built-in method buil
tins.sorted}
        1    0.042    0.042    0.042    0.042 {method 'randint' of
'mtrand.RandomState' objects}
        1    0.035    0.035    4.959    4.959 <string>:1(<module>)
        1    0.000    0.000    4.923    4.923 <ipython-input-23-7e4
471985ba7>:11(function1)
        1    0.000    0.000    4.959    4.959 {built-in method buil
tins.exec}
        1    0.000    0.000    0.000    0.000 {method 'disable' of
'_lsprof.Profiler' objects}
```

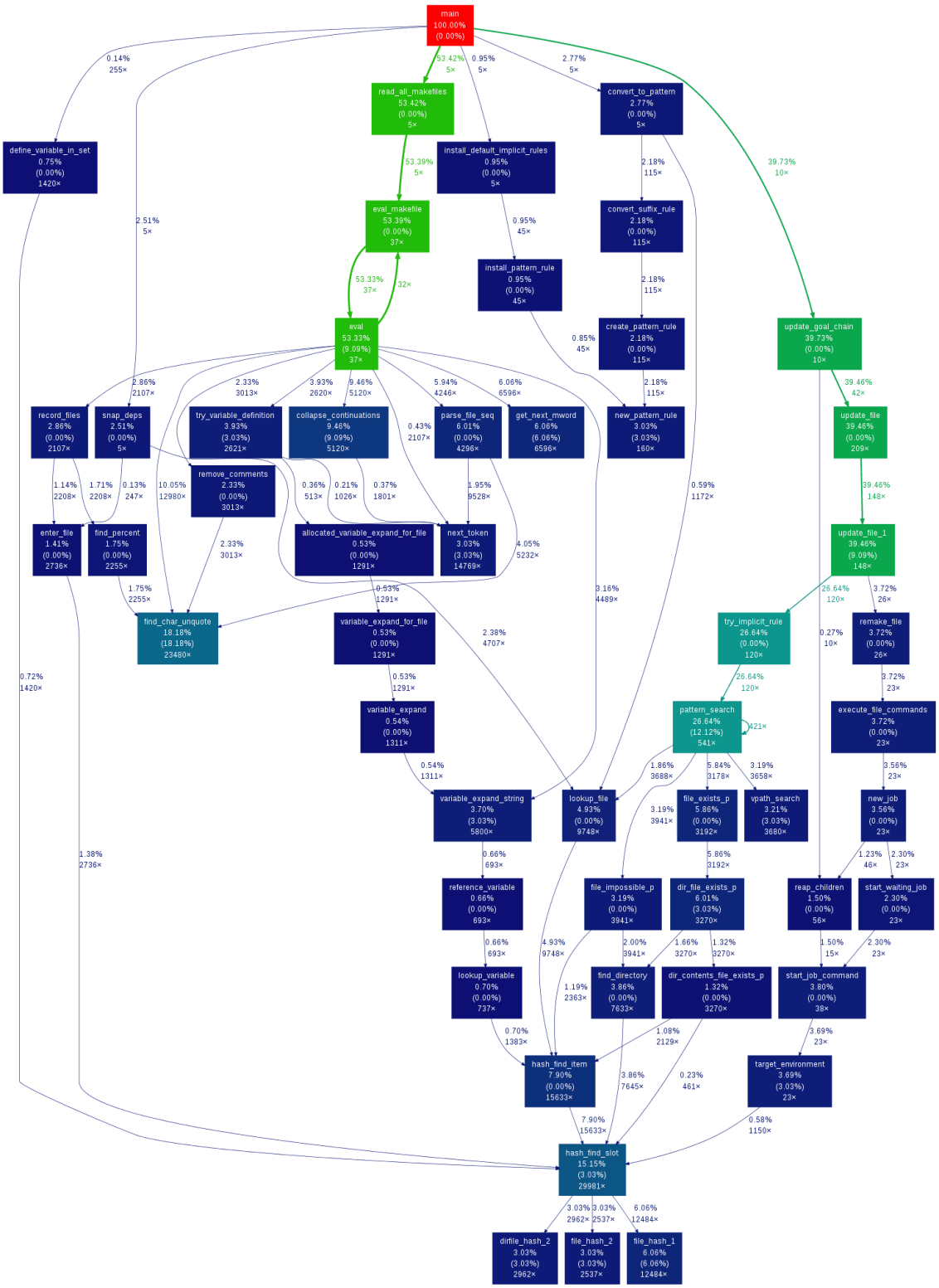
Or, for a beautifull call graph of a complex program:

```
   python3 -m cProfile -o profile.pstats script.py
      gprof2dot -f pstats profile.pstats | dot -Tpng -o profile.png
```

In [24]:

```
from IPython.display import Image
Image(filename='images/profile.png')
```

Out[24]:



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

In [25]:

```
def init_copy(f, g):
    """
    Just to be sure that the arguments are constant
    """
    return f.copy(), g.copy()
```

In [26]:

```
def finish(res):
    """
    A dummy function that does nothing
    """
    return res
```

In [27]:

```
import textwrap

def checker(ref, res):
    """
    A function that, given two results, check that they are identical
    """
    if (type(ref) is not type(res) or
        ref.dtype != res.dtype or
        not np.array_equal(res, ref)):

        print("Failed")

        print("types: ",type(ref), type(res))
        print("dtypes: ",ref.dtype, res.dtype)

        differ = np.where(np.logical_not(np.isclose(ref,res)))

        print(textwrap.dedent("""results:
                                   ref shape: {}
                                   res shape: {}

                                   idx
                                 {}

                                   ref
                                 {}

                                   res
                                 {}""".format(ref.shape,
                                              res.shape,
                                              differ,
                                              ref[differ],
                                              res[differ])
                              ))
        return False
    return True
```

In [28]:

```
def instrument(check=None, setup=init_copy, finish=finish, timing=True):
    """
    A decorator that will time a function, and if given,
    check it's result with a reference function

    setup and finish are part of the function not timed
    """
    def _check(function):
        def wrapped(*arg, check=check, timing=timing):

            # Our result
            a, b = setup(*arg)
            res = function(a, b)
            res = finish(res)

            if check is not None:
                print("Testing ", function.__name__, " ... ",end="")

                # The reference (might not be decorated)
                try:
                    ref = check(*arg, check=None, timing=False)
                except TypeError:
                    ref = check(*arg)

                if not checker(ref, res):
                    raise RuntimeError
                else:
                    print("OK")

            if timing:
                print("Timing  ", function.__name__, " ...")
                %timeit function(a, b)
            return res
        return wrapped
    return _check
```

In [29]:

```
# 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))
```

Python code (reference)

In [30]:

```
@instrument()
def py_simple_operations_with_tmparrays_and_loops(a, b):
    n = len(a)

    c = np.empty_like(a)
    for i in range(n):
        c[i] = a[i] * b[i]

    d = np.empty_like(a)
    for i in range(n):
        d[i] = 4.1 * a[i]

    e = np.empty_like(a)
    for i in range(n):
        e[i] = c[i] - d[i]

    f = np.empty_like(a)
    for i in range(n):
        f[i] = 2.5 * b[i]

    g = np.empty_like(a, dtype=np.bool)
    for i in range(n):
        g[i] = e[i] > f[i]

    return g

py_simple_operations_with_tmparrays_and_loops(a, b);
```

```
Timing   py_simple_operations_with_tmparrays_and_loops  ...
1.04 s ± 3.65 ms per loop (mean ± std. dev. of 7 runs, 1 loop each)
```

In [31]:

```
@instrument(check=py_simple_operations_with_tmparrays_and_loops)
def py_simple_operations_with_loops(a, b):
    n = len(a)

    g = np.empty_like(a, dtype=np.bool)
    for i in range(n):
        c = a[i] * b[i]
        d = 4.1 * a[i]
        e = c - d
        f = 2.5 * b[i]
        g[i] = e > f

    return g
py_simple_operations_with_loops(a, b);
```

```
Testing  py_simple_operations_with_loops  ... OK
Timing   py_simple_operations_with_loops  ...
578 ms ± 7.91 ms per loop (mean ± std. dev. of 7 runs, 1 loop each)
```

In [32]:

```
@instrument(check=py_simple_operations_with_loops)
def py_simple_operations(a, b):
    return a * b - 4.1 * a > 2.5 * b

py_simple_operations(a, b);
```

```
Testing  py_simple_operations  ... OK
Timing   py_simple_operations  ...
4.67 ms ± 135 µs per loop (mean ± std. dev. of 7 runs, 100 loops ea
ch)
```

In [33]:

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

i_py_tough_operations = instrument()(py_tough_operations)
i_py_tough_operations(a, b);
```

```
Timing   py_tough_operations  ...
58.8 ms ± 1.63 ms per loop (mean ± std. dev. of 7 runs, 10 loops ea
ch)
```

We can look at the bytecode (halfway to assembly)

And use it to optimize some things

In [34]:

```
import dis
print(dis.code_info(py_tough_operations))
print()
print("Code :")
dis.dis(py_tough_operations)
print()
```

```
Name:              py_tough_operations
Filename:          <ipython-input-33-918912a11f28>
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 :
  2           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 [35]:

```
from numpy import sin, arcsinh

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

i_py_tough_operations_with_localsin = instrument()(py_tough_operations_with_loca
lsin)
i_py_tough_operations_with_localsin(a, b);
```

```
Timing   py_tough_operations_with_localsin  ...
58.6 ms ± 1.09 ms per loop (mean ± std. dev. of 7 runs, 10 loops ea
ch)
```

In [36]:

```
import dis
print(dis.code_info(py_tough_operations_with_localsin))
print()
print("Code :")
dis.dis(py_tough_operations_with_localsin)
print()
```

```
Name:              py_tough_operations_with_localsin
Filename:          <ipython-input-35-bff33f479860>
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 [37]:

```
from numpy import sum as npsum

@instrument()
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, wmax = f.shape
    smax, tmax = g.shape

    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):

            v1 = x - smid
            v2 = v1 + smax
            w1 = y - tmid
            w2 = w1 + tmax

            h[x, y] = npsum(g * f[v1:v2, w1:w2])

    return h

convolve_python(small_f, g);
```

```
Timing   convolve_python  ...
169 ms ± 3.06 ms per loop (mean ± std. dev. of 7 runs, 10 loops eac
h)
```

When possible use already available function

In [38]:

```
from scipy.signal import convolve2d

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

@instrument(check=convolve_python, setup=scipy_setup)
def scipy_convolve(f, g):

    vmax, wmax = f.shape
    smax, tmax = g.shape

    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
```

In [39]:

```
scipy_convolve(small_f, g);
scipy_convolve(large_f, g, check=None);
```

```
Testing  scipy_convolve  ... OK
Timing   scipy_convolve  ...
6.19 ms ± 147 µs per loop (mean ± std. dev. of 7 runs, 100 loops ea
ch)
Timing   scipy_convolve  ...
693 ms ± 21 ms per loop (mean ± std. dev. of 7 runs, 1 loop each)
```

numexpr allows compilation of very simple code

And is multithreaded

In [40]:

```
import numexpr as ne

@instrument(check=py_simple_operations)
def ne_simple_operations(a, b):
    return ne.evaluate('a * b - 4.1 * a > 2.5 * b')

ne_simple_operations(a,b);
```

```
Testing  ne_simple_operations  ... OK
Timing   ne_simple_operations  ...
1.02 ms ± 34.9 µs per loop (mean ± std. dev. of 7 runs, 1000 loops
each)
```

In [41]:

```
@instrument(check=py_tough_operations_with_localsin)
def ne_tough_operations(a, b):
    return ne.evaluate("sin(a) + arcsinh(a / b)")

ne_tough_operations(a,b);
```

```
Testing  ne_tough_operations  ... OK
Timing   ne_tough_operations  ...
11.7 ms ± 62.2 µs per loop (mean ± std. dev. of 7 runs, 100 loops e
ach)
```

Numba allows compilation of more complex code using only decorators

And can parallelize part of your code

But it doesn't work everytime

In [42]:

```
import numba as nb

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

nb_simple_operations(a, b);
```

```
Testing  nb_simple_operations  ... OK
Timing   nb_simple_operations  ...
793 µs ± 4.71 µs per loop (mean ± std. dev. of 7 runs, 1000 loops e
ach)
```

In [43]:

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

nb_tough_operations(a, b);
```

```
Testing  nb_tough_operations  ... OK
Timing   nb_tough_operations  ...
14.2 ms ± 300 µs per loop (mean ± std. dev. of 7 runs, 100 loops ea
ch)
```

In [44]:

```
@instrument(check=scipy_convolve)
@nb.jit(nopython=True, nogil=True, cache=False, parallel=True)
def convolve_numba(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, wmax = f.shape
    smax, tmax = g.shape

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

    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
    return h

convolve_numba(small_f, g);
convolve_numba(large_f, g);
```

```
Testing  convolve_numba  ... OK
Timing   convolve_numba  ...
2.34 ms ± 71.8 µs per loop (mean ± std. dev. of 7 runs, 100 loops e
ach)
Testing  convolve_numba  ... OK
Timing   convolve_numba  ...
253 ms ± 5.73 ms per loop (mean ± std. dev. of 7 runs, 1 loop each)
```

In [45]:

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

    smax, tmax = g.shape

    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

@instrument(check=convolve_numba)
@nb.jit(nopython=True, nogil=True, cache=False, parallel=True)
def convolve_numba_with_stencil(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).astype(f.dtype)

convolve_numba_with_stencil(small_f, g);
convolve_numba_with_stencil(large_f, g);
```

```
Testing  convolve_numba_with_stencil  ... OK
Timing   convolve_numba_with_stencil  ...
1.5 ms ± 14 µs per loop (mean ± std. dev. of 7 runs, 1000 loops eac
h)
Testing  convolve_numba_with_stencil  ... OK
Timing   convolve_numba_with_stencil  ...
103 ms ± 2.51 ms per loop (mean ± std. dev. of 7 runs, 10 loops eac
h)
```

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 [46]:

```
%load_ext Cython
%set_env CFLAGS="-Ofast -march=native -fvect-cost-model=cheap" \
                "-fopenmp -Wno-unused-variable -Wno-cpp -Wno-maybe-uninitialize
d"
```

```
env: CFLAGS="-Ofast -march=native -fvect-cost-model=cheap"
        "-fopenmp -Wno-unused-variable -Wno-cpp -Wno-maybe-uninitia
lized"
```

In [47]:

```
%%cython
# cython: language_level=3
# cython: initializedcheck=False
# cython: binding=True
# cython: nonecheck=False
# distutils: extra_link_args = -fopenmp

import numpy as np
cimport numpy as np
cimport cython
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 functio
n
def csimple_operations(const double[::1]& a, const 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)
```

In [48]:

```
i_csimple_operations = instrument(check=nb_simple_operations)(csimple_operations
)
i_csimple_operations(a, b);
```

```
Testing  csimple_operations  ... OK
Timing   csimple_operations  ...
2.36 ms ± 639 µs per loop (mean ± std. dev. of 7 runs, 100 loops ea
ch)
```

In [49]:

```
%%cython
# cython: language_level=3
# cython: initializedcheck=False
# cython: binding=True
# cython: nonecheck=False
# cython: boundscheck=False
# cython: wraparound=False
# distutils: extra_link_args = -fopenmp

import numpy as np
cimport numpy as np
from libc.math cimport sin, asinh
from cython.parallel cimport parallel, prange

def ctough_operations(const double[::1]& a, const 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)
```

In [50]:

```
i_ctough_operations = instrument(check=nb_tough_operations)(ctough_operations)
i_ctough_operations(a, b);
```

```
Testing  ctough_operations  ... OK
Timing   ctough_operations  ...
15 ms ± 312 µs per loop (mean ± std. dev. of 7 runs, 100 loops eac
h)
```

In [51]:

```
%%cython
# cython: language_level=3
# cython: initializedcheck=False
# cython: binding=True
# cython: nonecheck=False
# cython: boundscheck=False
# cython: wraparound=False
# distutils: extra_link_args = -fopenmp

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

def convolve_cython(const long[:,::1]& f, const 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 long 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)
```

In [52]:

```
i_convolve_cython = instrument(check=convolve_numba_with_stencil)(convolve_cytho
n)
i_convolve_cython(small_f, g);
i_convolve_cython(large_f, g);
```

```
Testing  convolve_cython  ... OK
Timing   convolve_cython  ...
748 µs ± 12.5 µs per loop (mean ± std. dev. of 7 runs, 1000 loops e
ach)
Testing  convolve_cython  ... OK
Timing   convolve_cython  ...
81.2 ms ± 4.07 ms per loop (mean ± std. dev. of 7 runs, 10 loops ea
ch)
```

In [53]:

```
%%writefile CModule.h
#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);
```

```
Overwriting CModule.h
```

In [54]:

```
%%writefile CModule.c
#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;
        }
    }
}
```

```
Overwriting CModule.c
```

In [55]:

```
%%cython
# cython: language_level=3
# cython: initializedcheck=False
# cython: binding=True
# cython: nonecheck=False
# cython: boundscheck=False
# cython: wraparound=False
# distutils: extra_link_args = -fopenmp
# distutils: sources = CModule.c

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

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

def convolve_cython_pure(const long[:,::1]& f, const 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)
```

In [56]:

```
i_convolve_cython_pure = instrument(check=i_convolve_cython)(convolve_cython_pur
e)
i_convolve_cython_pure(small_f, g);
i_convolve_cython_pure(large_f, g);
```

```
Testing  convolve_cython_pure  ... OK
Timing   convolve_cython_pure  ...
731 µs ± 131 µs per loop (mean ± std. dev. of 7 runs, 1000 loops ea
ch)
Testing  convolve_cython_pure  ... OK
Timing   convolve_cython_pure  ...
64.8 ms ± 4.5 ms per loop (mean ± std. dev. of 7 runs, 10 loops eac
h)
```

In [57]:

```
!gcc CModule.c main.c -o full_c -Ofast -march=native -fopenmp -fvect-cost-model=
cheap
!./full_c
```

```
Time :    0s   57ms  267us  239ns
```

Fortran through f2py is also very efficient

But you have to

- rewrite your code
- be carefull with memory organization
- compile it separately

In [58]:

```
%load_ext fortranmagic
%set_env F90FLAGS="-Ofast -march=native -fvect-cost-model=cheap -fopenmp"
%fortran_config --extra="-lgomp"  --extra="--noarch"
```

```
/Data/WORK/Formations/Python/formation_python/venv/lib/python3.5/si
te-packages/fortranmagic.py:147: UserWarning: get_ipython_cache_dir
has moved to the IPython.paths module since IPython 4.0.
  self._lib_dir = os.path.join(get_ipython_cache_dir(), 'fortran')
```

```
env: F90FLAGS="-Ofast -march=native -fvect-cost-model=cheap -fopenm
p"
New default arguments for %fortran:
        --extra="-lgomp"  --extra="--noarch"
```

In [59]:

```
%%fortran
subroutine fsimple_operations(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 fsimple_operations
```

In [60]:

```
@instrument(check=i_csimple_operations)
def i_fsimple_operations(a ,b):
    return fsimple_operations(a, b).astype(bool)

i_fsimple_operations(a, b);
```

```
Testing  i_fsimple_operations  ... OK
Timing   i_fsimple_operations  ...
1.28 ms ± 30.3 µs per loop (mean ± std. dev. of 7 runs, 1000 loops
each)
```

In [61]:

```
%%fortran
subroutine ftough_operations(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 ftough_operations
```

In [62]:

```
@instrument(check=i_ctough_operations)
def i_ftough_operations(a ,b):
    return ftough_operations(a, b)

i_ftough_operations(a, b);
```

```
Testing  i_ftough_operations  ... OK
Timing   i_ftough_operations  ...
12 ms ± 985 µs per loop (mean ± std. dev. of 7 runs, 100 loops eac
h)
```

In [63]:

```
%%fortran
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) :: x, y
    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.

            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
```

In [64]:

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

@instrument(check=i_convolve_cython_pure, setup=fortran_setup)
def i_convolve_fortran(f, g):
    h, err = convolve_fortran(f, g)
    if err:
        print(err)
        raise ValueError("FORTRAN ERROR ! (Probably : Only odd dimensions on fil
ter supported)")
    return h

i_convolve_fortran(small_f, g);
i_convolve_fortran(large_f, g);
```

```
Testing  i_convolve_fortran  ... OK
Timing   i_convolve_fortran  ...
707 µs ± 86.6 µs per loop (mean ± std. dev. of 7 runs, 1000 loops e
ach)
Testing  i_convolve_fortran  ... OK
Timing   i_convolve_fortran  ...
70 ms ± 3.31 ms per loop (mean ± std. dev. of 7 runs, 10 loops eac
h)
```

In [65]:

```
%%writefile fortranModule.f90
module fortranmodule
    implicit none
    contains

    subroutine convolve_fortran_pure(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) :: x, y
        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.

                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_pure
end module fortranmodule
```

```
Overwriting fortranModule.f90
```

In [66]:

```
!gfortran fortranModule.f90 main.f90 -o full_f  -Ofast -march=native -fopenmp -f
vect-cost-model=cheap
!./full_f
```

```
Time :   0s   61ms 930us 281ns
```

- parallelism
  - cuda

In [67]:

```
"""
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 - tmid)];

    }
    }
}
""")
```

In [68]:

```
"""
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)
```

In [70]:

```
"""
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_bl
ock,
                                        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", "-defaul
t-stream=per-thread"])
print("Compilation OK")

__convolve_cuda = cuda_module.get_function('convolve_cuda')

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

```
Compilation OK
```

In [75]:

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

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()

@instrument(check=i_convolve_cython_pure)
def convolve_cuda(f, g):
    f_gpu, g_gpu = cuda_setup(f, g)
    h_gpu = gpuarray.zeros_like(f_gpu)
    _convolve_cuda(f_gpu, g_gpu, h_gpu)
    return cuda_finish(h_gpu)

convolve_cuda(large_f, g);
```

```
Testing  convolve_cuda  ... OK
Timing   convolve_cuda  ...
42.1 ms ± 265 µs per loop (mean ± std. dev. of 7 runs, 10 loops eac
h)
```

In [76]:

```
"""
CUDA DOESN'T WORK ON THE VIRTUAL MACHINE
YOU ARE WELCOME TO TRY THIS ON YOU OWN COMPUTER
"""
@instrument(check=i_convolve_cython_pure, setup=cuda_setup, finish=cuda_finish)
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

convolve_cuda2(large_f, g);
```

```
Testing  convolve_cuda2  ... OK
Timing   convolve_cuda2  ...
31.9 ms ± 189 µs per loop (mean ± std. dev. of 7 runs, 10 loops eac
h)
```

Conclusion on optimisation

(values may be different as before)

Simple operations

context	time	comment
Python	1040ms	naive implementation
Python	578ms	removing tmparrays
Python	4.67ms	using implicit loops
numexpr	1.02ms	
numba	**793us**	
cython	2.36ms	
f2py	1.28ms	

Tough operations

context	time	comment
Python	58.8ms	naive implementation
Python	58.6ms	using local sin
numexpr	**11.7ms**	
numba	14.2ms	
cython	15ms	
f2py	12ms	

Convolution

context	time small case	time large case	comment
Python	169ms		naive implementation
scipy	6.19ms	693ms	
numba	2.34ms	253ms	
numba	1.50ms	103ms	using stencil
cython	748us	81.2ms	using numpy datastructure
cython	731us	64.8ms	using c datastructure
c		57.2ms	
f2py	**707ms**	70ms	
fortran		61.9ms	
cuda		42.1ms	including communication
cuda		**31.9ms**	excluding communication

In [77]:

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

- asyncio
  - not a real parallelism
  - effective for io-bound tasks (web)
  - not very interesting here

- joblib
  - real parallelism
  - limited to one computer
  - relatively easy to use
  - multithreading of multiprocessing

In [78]:

```
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())
```

```
12.734344482421875
100.37757168265287
```

- multithreading
  - real parallelism
  - limited to one computer
  - shared memory

In [79]:

```
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())
```

```
13.602105617523193
98.02541523722961
```

- multiprocessing
  - real parallelism
  - limited to one computer

In [80]:

```
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(npro
cs)]

    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())
```

```
12.50019359588623
98.93560450067852
```

- mpi (mpi4py)
  - real parallelism
  - unlimited
  - relatively complex to use (same as in C, fortran, ...)

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.

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 ± 262 µs per loop (mean ± std. dev. of 7 runs, 100 loops each)
Computing
503 µs ± 5.41 µs per loop (mean ± std. dev. of 7 runs, 1000 loops each)
```

My solution

```
Reading Files
65.3 µs ± 1.58 µs per loop (mean ± std. dev. of 7 runs, 10000 loops each)
Computing
66.2 µs ± 1.02 µs per loop (mean ± std. dev. of 7 runs, 10000 loops each)
```

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

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

In [81]:

```
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
```

In [82]:

```
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 wrapped
    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)
```

In [83]:

```
def read_values(filename, description):
    """
    Read all the values
    """
    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
```

In [84]:

```
def read_file(filename):
    """
    Read an image in the PGM format
    """
    # read the header part
    description = get_description(filename)

    # read the values
    values = read_values(filename, description)
    return values
```

In [85]:

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

    return data
```

In [86]:

```
def get_neighbors(tab, i, j):
    """
    Extract from the array the neighbors of a pixel
    """
    neigh = []
    for jrange in [-1, 0, 1]:
        for irange in [-1, 0, 1]:
            neigh.append(tab[i + irange][j + jrange])
    return neigh
```

In [87]:

```
import math

def compute_wtf(neigh):
    """
    Apply a reduction operation on the array neigh
    """
    value = 1.
    for i in range(len(neigh)):
        value *= math.exp(neigh[i]) ** (1 / len(neigh))
    value = math.log(value)

    return float(value)
```

In [88]:

```
def kernel(tab):
    """
    Apply compute_wtf on each pixel except boundary
    """
    xdim = len(tab)
    ydim = len(tab[0])

    # create the result list
    result = []

    #1st line contains only 0
    result.append([0])
    for jrange in range(1, ydim):
        result[0].append(0)

    for irange in range(1, xdim - 1):
        #1st column contains only 0
        result.append([0])

        # For each pixel inside the image
        for jrange in range(1, ydim - 1):
            # Extract the neighboring pixels
            neigh = get_neighbors(tab, irange, jrange)

            # Apply compute_wtf on it
            res = compute_wtf(neigh)

            # Store the result
            result[irange].append(res)

        #last colum contains only 0
        result[irange].append(0)

    #last line contains only 0
    result.append([])
    for jrange in range(ydim):
        result[xdim - 1].append(0)

    return result
```

In [89]:

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

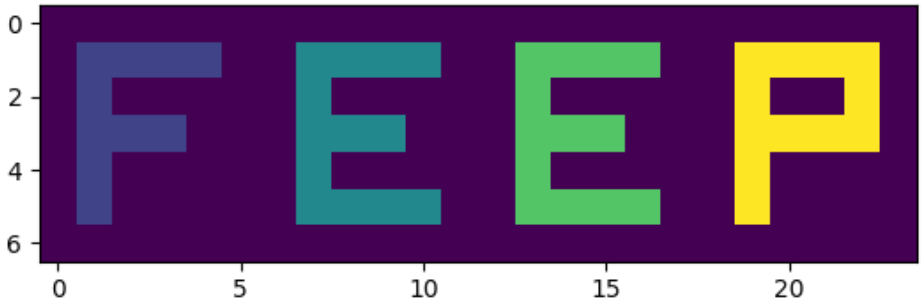
In [90]:

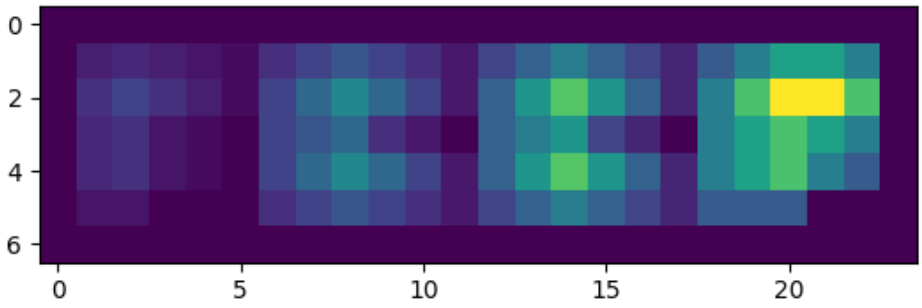
```
%matplotlib notebook
import matplotlib.pyplot as plt

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()
```

In [91]:

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

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

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

    plot(data)
    plot(result)
```

```
Reading Files
7.31 ms ± 242 µs per loop (mean ± std. dev. of 7 runs, 100 loops ea
ch)
Computing
509 µs ± 8.25 µs per loop (mean ± std. dev. of 7 runs, 1000 loops e
ach)
```





Solution

In [92]:

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

    if file is an opened file:
        go back the the begining of the file
        read the header
        leave the file at the end of header (start of values)
    else:
        call itself with the file openened
        (this should be never called)
    """
    if isinstance(file, str):
        with open(file,'r', encoding="utf-8") as opened_file:
            return get_description(opened_file)

    # 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
```

In [93]:

```
def read_values(file, description):
    """
    Read all the values directly
    The file must be already opened
    The values are stored in a numpy array
    """
    # 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] = vals
        i += nvals
    return values.reshape((nx, ny))
```

In [94]:

```
def read_file(filename):
    """
    Read an image in the PGM format

    Open the file *once*
    """
    # 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
```

In [95]:

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

    return data
```

In [96]:

```
%load_ext Cython
%set_env CFLAGS="-Ofast -march=native -fvect-cost-model=cheap -fopenmp -Wno-unus
ed-variable -Wno-cpp -Wno-maybe-uninitialized"
```

```
The Cython extension is already loaded. To reload it, use:
  %reload_ext Cython
env: CFLAGS="-Ofast -march=native -fvect-cost-model=cheap -fopenmp
-Wno-unused-variable -Wno-cpp -Wno-maybe-uninitialized"
```

In [97]:

```
%%cython --link-args=-fopenmp
# cython: language_level=3
# cython: initializedcheck=False
# cython: binding=True
# cython: nonecheck=False
# cython: boundscheck=False
# cython: wraparound=False
# distutils: extra_link_args = -fopenmp

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

def ckernel(const double[:,::1] &data, const 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 np.asarray(res)
```

In [98]:

```
def kernel(i):
    """
    Apply compute_wtf on each pixel except boundary
    """
    global data, result, nt_omp, nt_job
    if data[i].size < 1000:
        result[i] = ckernel(data[i], 1)
    else:
        result[i] = ckernel(data[i], nt_job)
```

In [99]:

```
import os
from threading import Thread, RLock

nprocs = os.cpu_count()

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)
```

In [100]:

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

    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()
```

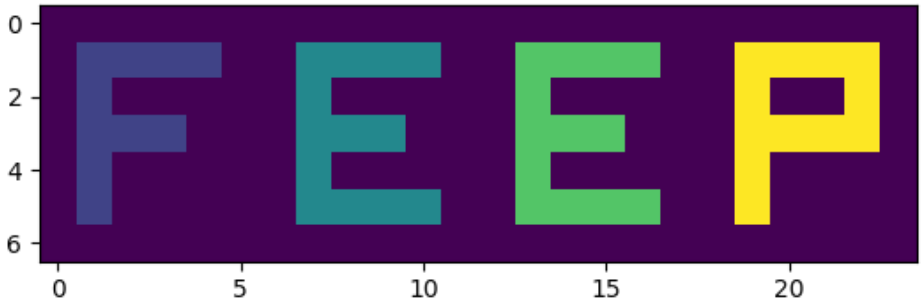
In [101]:

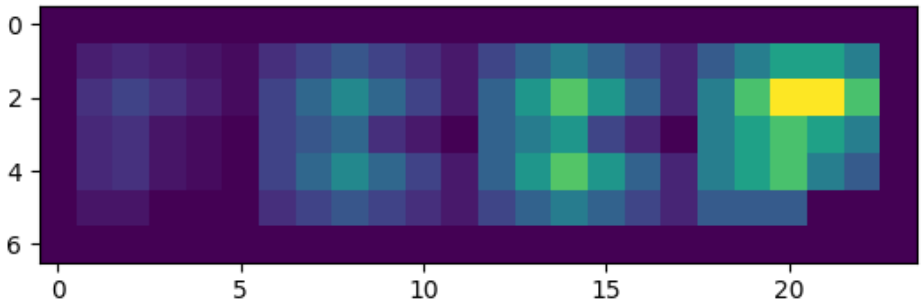
```
%matplotlib notebook
import matplotlib.pyplot as plt

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()
```

In [102]:

```
from copy import deepcopy

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

nt_job = min(nprocs // 2, len(files))
nt_omp = nprocs - nt_job

if __name__ == "__main__":

    print("Reading Files")
    data = init(files)
    %timeit init(files)

    print("Computing")
    #sort data: biggest images first for better equilibrium in parallel
    data = sorted(data, key=np.size, reverse=True)

    #prepare result array
    result = deepcopy(data)

    job(data)
    %timeit job(data)

    plot(data)
    plot(result)
```

```
Reading Files
61.8 µs ± 1.17 µs per loop (mean ± std. dev. of 7 runs, 10000 loops
each)
Computing
53.9 µs ± 1.16 µs per loop (mean ± std. dev. of 7 runs, 10000 loops
each)
```




0
2
4
6
0
5
10
15
20