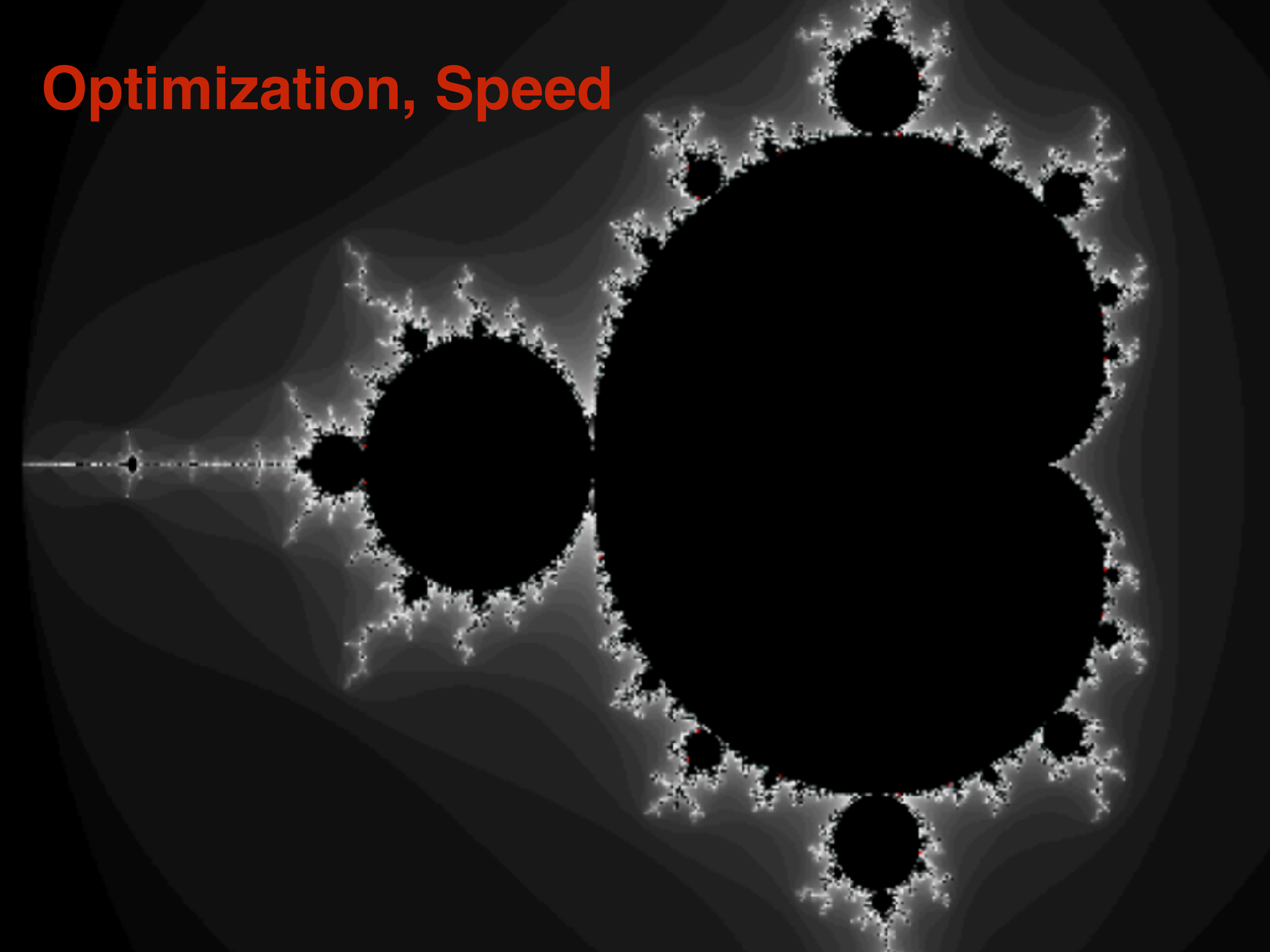
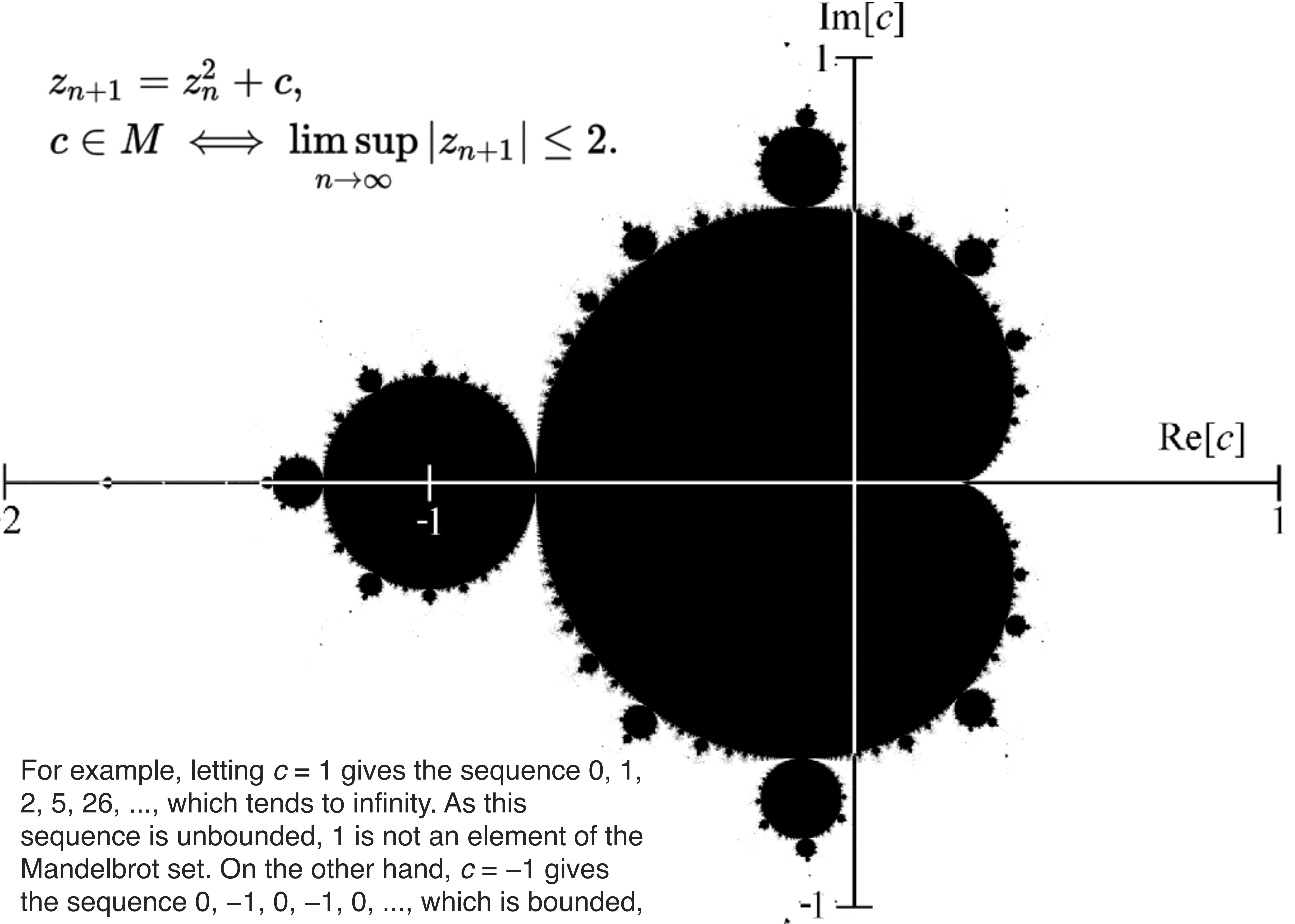
# Optimization, Speed



$$z_{n+1} = z_n^2 + c,$$
$$c \in M \iff \limsup_{n\to\infty} |z_{n+1}| \le 2.$$




For example, letting $c = 1$ gives the sequence 0, 1, 2, 5, 26, ..., which tends to infinity. As this sequence is unbounded, 1 is not an element of the Mandelbrot set. On the other hand, $c = -1$ gives the sequence 0, –1, 0, –1, 0, ..., which is bounded, and so –1 belongs to the Mandelbrot set.

The Mandelbrot set is the set of values of $c$ in the complex plane for which the orbit of 0 under iteration of the quadratic map

$$z_{n+1} = z_n^2 + c$$

remains bounded.[13] That is, a complex number $c$ is part of the Mandelbrot set if, when starting with $z_0 = 0$ and applying the iteration repeatedly, the absolute value of $z_n$ remains bounded however large $n$ gets. This can also be represented as[14]

$$z_{n+1} = z_n^2 + c,$$
$$c \in M \iff \limsup_{n\to\infty} |z_{n+1}| \leq 2.$$

For example, letting $c = 1$ gives the sequence 0, 1, 2, 5, 26, ..., which tends to infinity. As this sequence is unbounded, 1 is not an element of the Mandelbrot set. On the other hand, $c = -1$ gives the sequence 0, −1, 0, −1, 0, ..., which is bounded, and so −1 belongs to the Mandelbrot set.

The Mandelbrot set $M$ is defined by a family of complex quadratic polynomials

$$P_c : \mathbb{C} \to \mathbb{C}$$

given by

$$P_c : z \mapsto z^2 + c,$$

where $c$ is a complex parameter. For each $c$, one considers the behavior of the sequence

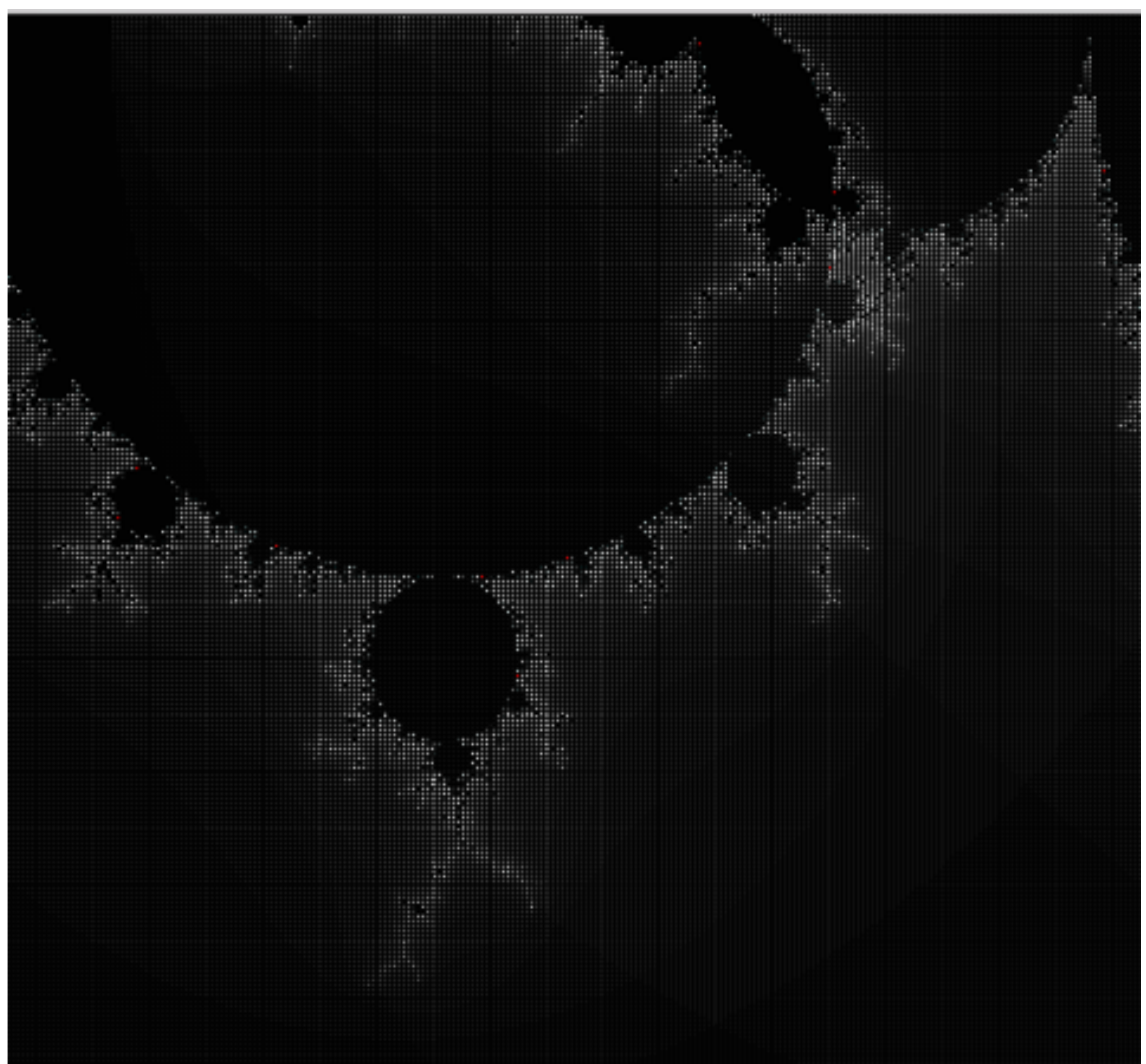
$$(0, P_c(0), P_c(P_c(0)), P_c(P_c(P_c(0))), \ldots)$$

pure python code

```
Main took 0:00:12.465387    python pure_python.py 1000 1000
Main took 0:00:01.190245    pypy pure_python.py 1000 1000
```

pure python code
better referencing

```
Main took 0:00:08.769609    python pure_python_2.py 1000 1000
Main took 0:00:00.863155    pypy pure_python_2.py 1000 1000
```



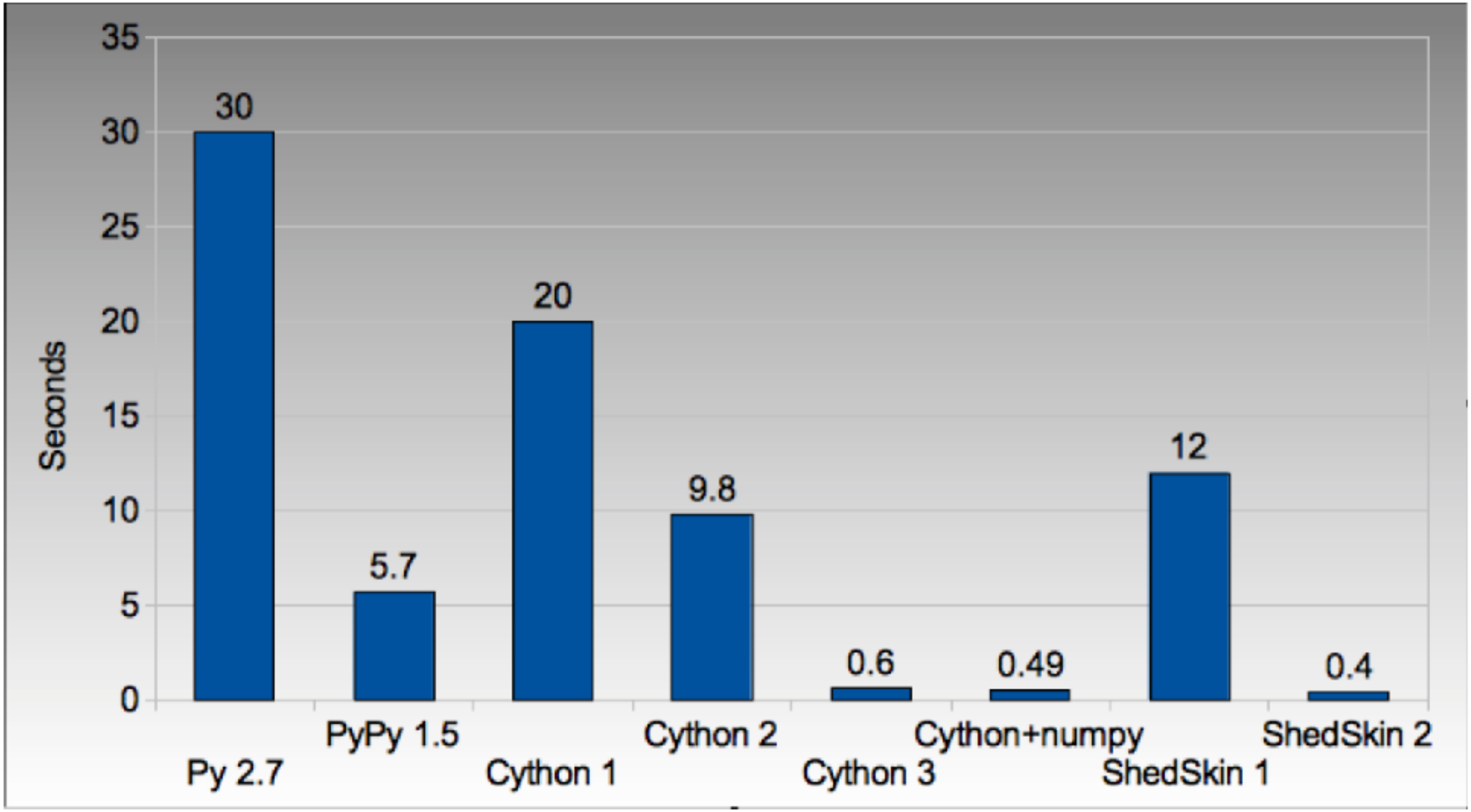



Figure 4.1: Run times on laptop for Python/C implementations

The profile module is the standard way to profile Python code, take a look at it here http://docs.python.org/library/profile.html. We'll run it on our simple Python implementation:

```
python -m cProfile -o rep.prof pure_python.py 1000 1000
```

This generates a rep.prof output file containing the profiling results, we can now load this into the pstats module and print out the top 10 slowest functions:

```
import pstats
p = pstats.Stats('rep.prof')
p.sort_stats('cumulative').print_stats(10)
```

```
import pstats
p = pstats.Stats('rep.prof')
p.sort_stats('cumulative').print_stats(10)
```

```
Thu Feb 19 08:11:49 2015    rep.prof

         51927850 function calls (51927727 primitive calls) in 18.920 seconds

   Ordered by: cumulative time
   List reduced from 656 to 10 due to restriction <10>

   ncalls  tottime  percall  cumtime  percall filename:lineno(function)
        1    0.011    0.011   18.920   18.920 pure_python.py:1(<module>)
        1    0.077    0.077   18.909   18.909 pure_python.py:23(calc_pure_python)
        1   14.250   14.250   18.610   18.610 pure_python.py:9(calculate_z_serial_purepython)
 51414419    3.366    0.000    3.366    0.000 {abs}
   250076    0.994    0.000    0.994    0.000 {range}
        1    0.008    0.008    0.154    0.154 /usr/local/lib/python2.7/site-packages/numpy/__init__.py:106(<mo
        1    0.000    0.000    0.118    0.118 /usr/local/lib/python2.7/site-packages/numpy/add_newdocs.py:10(<
        1    0.009    0.009    0.116    0.116 /usr/local/lib/python2.7/site-packages/numpy/lib/__init__.py:1(<
        1    0.001    0.001    0.092    0.092 /usr/local/lib/python2.7/site-packages/numpy/lib/type_check.py:3
        1    0.017    0.017    0.091    0.091 /usr/local/lib/python2.7/site-packages/numpy/core/__init__.py:1(


<pstats.Stats instance at 0x110a2b518>
```

*However* - which *lines* are causing our code to run slow? This is the more interesting question and `cProfile` can't answer it.

Let's look at the `line_profiler` module. First we have to decorate our chosen function with `@profile`:

```
@profile
def calculate_z_serial_purepython(q, maxiter, z):
```

Next we'll run `kernprof.py` and ask it to do line-by-line profiling and to give us a visual output, then we tell it what to profile. **Note** that we're running a much smaller problem as line-by-line profiling takes ages:

```
>> kernprof.py -l -v pure_python.py 300 100
```

```
pip install line_profiler
```

Excursion into Decorators: http://thecodeship.com/patterns/guide-to-python-function-decorators/

the code ship

# A guide to Python's function decorators

Python is rich with powerful features and expressive syntax. One of my favorites is decorators. In the context of design patterns, decorators dynamically alter the functionality of a function, method or class without having to directly use subclasses. This is ideal when you need to extend the functionality of functions that you don't want to modify. We can implement the decorator pattern anywhere, but Python facilitates the implementation by providing much more expressive features and syntax for that.

In this post I will be discussing Python's function decorators in depth, accompanied by a bunch of examples on the way to clear up the concepts. All examples are in Python 2.7 but the same concepts should apply to Python 3 with some change in the syntax.

Essentially, decorators work as wrappers, modifying the behavior of the code before and after a target function execution, without the need to modify the function itself, augmenting the original functionality, thus decorating it.

## Assign functions to variables

```
def greet(name):
    return "hello "+name

greet_someone = greet
print greet_someone("John")

# Outputs: hello John
```

## Define functions inside other functions

```
def greet(name):
    def get_message():
        return "Hello "

    result = get_message()+name
    return result

print greet("John")

# Outputs: Hello John
```

## Functions can be passed as parameters to other functions

```
def greet(name):
   return "Hello " + name

def call_func(func):
    other_name = "John"
    return func(other_name)

print call_func(greet)

# Outputs: Hello John
```

# Functions can return other functions

In other words, functions generating other functions.

```
def compose_greet_func():
    def get_message():
        return "Hello there!"

    return get_message

greet = compose_greet_func()
print greet()

# Outputs: Hello there!
```

## Inner functions have access to the enclosing scope

More commonly known as a closure. A very powerful pattern that we will come across while building decorators. Another thing to note, Python only allows read access to the outer scope and not assignment. Notice how we modified the example above to read a "name" argument from the enclosing scope of the inner function and return the new function.

```
def compose_greet_func(name):
    def get_message():
        return "Hello there "+name+"!"

    return get_message

greet = compose_greet_func("John")
print greet()

# Outputs: Hello there John!
```

## Composition of Decorators

Function decorators are simply wrappers to existing functions. Putting the ideas mentioned above together, we can build a decorator. In this example let's consider a function that wraps the string output of another function by **p** tags.

```
def get_text(name):
   return "lorem ipsum, {0} dolor sit amet".format(name)

def p_decorate(func):
   def func_wrapper(name):
       return "<p>{0}</p>".format(func(name))
   return func_wrapper

my_get_text = p_decorate(get_text)

print my_get_text("John")

# <p>Outputs lorem ipsum, John dolor sit amet</p>
```

That was our first decorator. A function that takes another function as an argument, generates a new function, augmenting the work of the original function, and returning the generated function so we can use it anywhere. To have get_text itself be decorated by p_decorate, we just have to assign get_text to the result of p_decorate.

```
get_text = p_decorate(get_text)

print get_text("John")

# Outputs lorem ipsum, John dolor sit amet
```

Another thing to notice is that our decorated function takes a name argument. All what we had to do in the decorator is to let the wrapper of get_text pass that argument.

## Python's Decorator Syntax

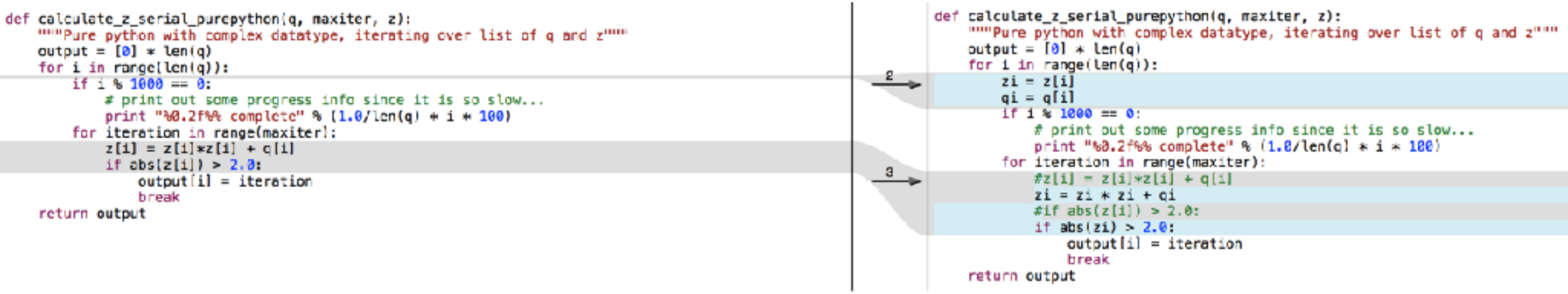
Python makes creating and using decorators a bit cleaner and nicer for the programmer through some syntactic sugar To decorate get_text we don't have to `get_text = p_decorator(get_text)` There is a neat shortcut for that, which is to mention the name of the decorating function before the function to be decorated. The name of the decorator should be perpended with an @ symbol.

```
def p_decorate(func):
   def func_wrapper(name):
       return "<p>{0}</p>".format(func(name))
   return func_wrapper

@p_decorate
def get_text(name):
   return "lorem ipsum, {0} dolor sit amet".format(name)

print get_text("John")

# Outputs <p>lorem ipsum, John dolor sit amet</p>
```

```
Total sum of elements (for validation): 75014
Wrote profile results to pure_python.py.lprof
Timer unit: 1e-06 s

Total time: 0.806372 s
File: pure_python.py
Function: calculate_z_serial_purepython at line 9

Line #      Hits         Time  Per Hit   % Time  Line Contents
==============================================================
     9                                           @profile
    10                                           def calculate_z_serial_purepython(q, maxiter,
    11                                               """Pure python with complex datatype, ite
    12         1          128    128.0      0.0      output = [0] * len(q)
    13     22501         9281      0.4      1.2      for i in range(len(q)):
    14     22500         9853      0.4      1.2          if i % 1000 == 0:
    15                                                       # print out some progress info si
    16        23          403     17.5      0.0              print "%0.2f%% complete" % (1.0/l
    17    560314       220829      0.4     27.4          for iteration in range(maxiter):
    18    555686       293387      0.5     36.4              z[i] = z[i]*z[i] + q[i]
    19    555686       255247      0.5     31.7              if abs(z[i]) > 2.0:
    20     17872         7377      0.4      0.9                  output[i] = iteration
    21     17872         9866      0.6      1.2                  break
    22         1            1      1.0      0.0      return output
```

```
18        z[i] = z[i]*z[i] + q[i]
19        if abs(z[i]) > 2.0:
```

```
>>> import pure_python # imports our solver into Python
>>> dis.dis(pure_python.calculate_z_serial_purepython)
....
 18          109 LOAD_FAST              2 (z)     # load z
             112 LOAD_FAST              4 (i)     # load i
             115 BINARY_SUBSCR                    # get value in z[i]
             116 LOAD_FAST              2 (z)     # load z
             119 LOAD_FAST              4 (i)     # load i
             122 BINARY_SUBSCR                    # get value in z[i]
             123 BINARY_MULTIPLY                  # z[i] * z[i]
             124 LOAD_FAST              0 (q)     # load z
             127 LOAD_FAST              4 (i)     # load i
             130 BINARY_SUBSCR                    # get q[i]
             131 BINARY_ADD                       # add q[i] to last multiply
             132 LOAD_FAST              2 (z)     # load z
             135 LOAD_FAST              4 (i)     # load i
             138 STORE_SUBSCR                     # store result in z[i]

 19          139 LOAD_GLOBAL            2 (abs)   # load abs function
             142 LOAD_FAST              2 (z)     # load z
             145 LOAD_FAST              4 (i)     # load i
             148 BINARY_SUBSCR                    # get z[i]
             149 CALL_FUNCTION          1         # call abs
             152 LOAD_CONST             6 (2.0)   # load 2.0
             155 COMPARE_OP             4 (>)     # compare result of abs with 2.0
             158 POP_JUMP_IF_FALSE    103         # jump depending on result
...
```




pure python code

```
def calculate_z_serial_purepython(q, maxiter, z):
    """Pure python with complex datatype, iterating over list of q and z"""
    output = [0] * len(q)
    for i in range(len(q)):
        if i % 1000 == 0:
            # print out some progress info since it is so slow...
            print "%0.2f%% complete" % (1.0/len(q) * i * 100)
        for iteration in range(maxiter):
            z[i] = z[i]*z[i] + q[i]
            if abs(z[i]) > 2.0:
                output[i] = iteration
                break
    return output
```

pure python code improved

```
def calculate_z_serial_purepython(q, maxiter, z):
    """Pure python with complex datatype, iterating over list of q and z"""
    output = [0] * len(q)
    for i in range(len(q)):
        zi = z[i]
        qi = q[i]
        if i % 1000 == 0:
            # print out some progress info since it is so slow...
            print "%0.2f%% complete" % (1.0/len(q) * i * 100)
        for iteration in range(maxiter):
            #z[i] = z[i]*z[i] + q[i]
            zi = zi * zi + qi
            #if abs(z[i]) > 2.0:
            if abs(zi) > 2.0:
                output[i] = iteration
                break
    return output
```

```
Total time: 0.804272 s
File: pure_python_2.py
Function: calculate_z_serial_purepython at line 10

Line #      Hits         Time  Per Hit   % Time  Line Contents
==============================================================
    10                                           @profile
    11                                           def calculate_z_serial_purepython(q, maxit
    12                                               """Pure python with complex datatype, i
    13         1          119    119.0      0.0      output = [0] * len(q)
    14     22501         9386      0.4      1.2      for i in range(len(q)):
    15     22500         9574      0.4      1.2          zi = z[i]
    16     22500         9512      0.4      1.2          qi = q[i]
    17     22500        10169      0.5      1.3          if i % 1000 == 0:
    18                                                       # print out some progress info
    19        23          437     19.0      0.1              print "%0.2f%% complete" % (1.0/
    20    560314       231067      0.4     28.7          for iteration in range(maxiter):
    21                                                       #z[i] = z[i]*z[i] + q[i]
    22    555686       257318      0.5     32.0              zi = zi * zi + qi
    23                                                       #if abs(z[i]) > 2.0:
    24    555686       258388      0.5     32.1              if abs(zi) > 2.0:
    25     17872         7872      0.4      1.0                  output[i] = iteration
    26     17872        10429      0.6      1.3                  break
    27         1            1      1.0      0.0      return output
```

```
20          for iteration in range(maxiter):
21              #z[i] = z[i]*z[i] + q[i]
22              zi = zi * zi + qi
23              #if abs(z[i]) > 2.0:
24              if abs(zi) > 2.0:
25                  output[i] = iteration
26                  break
```

```
        >>  123 FOR_ITER                52 (to 178)
            126 STORE_FAST               7 (iteration)

22          129 LOAD_FAST                5 (zi)
            132 LOAD_FAST                5 (zi)
            135 BINARY_MULTIPLY
            136 LOAD_FAST                6 (qi)
            139 BINARY_ADD
            140 STORE_FAST               5 (zi)

24          143 LOAD_GLOBAL              2 (abs)
            146 LOAD_FAST                5 (zi)
            149 CALL_FUNCTION            1
            152 LOAD_CONST               6 (2.0)
            155 COMPARE_OP               4 (>)
            158 POP_JUMP_IF_FALSE      123
```

```python
def calculate_z_numpy(q, maxiter, z):
    """use vector operations to update all zs and qs to create new output array"""
    output = np.resize(np.array(0,), q.shape)
    for iteration in range(maxiter):
        z = z*z + q
        done = np.greater(abs(z), 2.0)
        q = np.where(done,0+0j, q)
        z = np.where(done,0+0j, z)
        output = np.where(done, iteration, output)
    return output
```

`numpy`'s strength is that it simplifies running the same operation on a vector (or matrix) of numbers rather than on individual items in a `list` one at a time.

If your problem normally involves using nested `for` loops to iterate over individual items in a `list` then consider whether `numpy` could do the same job for you in a simpler (and probably faster) fashion.

If the above code looks odd to you, read it as:

- `z*z` does a pairwise multiplication, think of it as `z[0] = z[0] * z[0]; z[1] = z[1] * z[1]; ...; z[n-1] = z[n-1] * z[n-1]`.
- `z_result + q` does a pairwise addition, just like the line above but adding the result
- `z = ...` assigns the new array back to `z`
- `np.greater(condition, item_if_True, item_if_False)` calculates the condition for each item in `abs(z)`, for the nth value if the result is `True` it uses the `item_if_true` value (in this case `0+0j`) else it uses the other value (in this case `q[nth]`) - each item in `q` either resets to `0+0j` or stays at the value it was before
- The same thing happens for `z`
- `output`'s items are set to `iteration` if `done[nth] == True` else they stay at the value they were at previously.

```
>>>python numpy_vector.py 1000 1000
x and y have length: 500 500
Total elements: 250000
Main took 0:00:02.927419
Total sum of elements (for validation): 1148485

>>>python numpy_vector_2.py 1000 1000
x and y have length: 500 500
Total elements: 250000
STEP_SIZE 20000
Main took 0:00:02.488578
Total sum of elements (for validation): 1148485
```

```python
def calculate_z_numpy(q, maxiter, z):
    """use vector operations to update all zs and qs
    output = np.resize(np.array(0,), q.shape)
    for iteration in range(maxiter):
        z = z*z + q
        done = np.greater(abs(z), 2.0)
        q = np.where(done,0+0j, q)
        z = np.where(done,0+0j, z)
        output = np.where(done, iteration, output)
    return output
```

```python
def calculate_z_numpy(q_full, maxiter, z_full):
    output = np.resize(np.array(0,), q_full.shape)
    #STEP_SIZE = len(q_full) # 54s for 250,000
    #STEP_SIZE = 90000 # 52
    #STEP_SIZE = 50000 # 45s
    #STEP_SIZE = 45000 # 45s
    STEP_SIZE = 20000 # 42s # roughly this looks optimal on Ma
    #STEP_SIZE = 10000 # 43s
    #STEP_SIZE = 5000 # 45s
    #STEP_SIZE = 1000 # 1min02
    #STEP_SIZE = 100 # 3mins
    print "STEP_SIZE", STEP_SIZE
    for step in range(0, len(q_full), STEP_SIZE):
        z = z_full[step:step+STEP_SIZE]
        q = q_full[step:step+STEP_SIZE]
        for iteration in range(maxiter):
            z = z*z + q
            done = np.greater(abs(z), 2.0)
            q = np.where(done,0+0j, q)
            z = np.where(done,0+0j, z)
            output[step:step+STEP_SIZE] = np.where(done, itera
    return output
```

# MULTIPROCESSING

The `multiprocessing` module lets us send work units out as new Python processes on our local machine (it won't send jobs over a network). For jobs that require little or no interprocess communication it is ideal.

We need to split our input lists into shorter work lists which can be sent to the new processes, we'll then need to combine the results back into a single `output` list.

We have to split our `q` and `z` lists into shorter chunks, we'll make one sub-list per CPU. On my MacBook I have two cores so we'll split the 250,000 items into two 125,000 item lists. If you only have one CPU you can hard-code `nbr_chunks` to e.g. 2 or 4 to see the effect.

```
# create a Pool which will create Python processes
p = multiprocessing.Pool()
start_time = datetime.datetime.now()
# send out the work chunks to the Pool
# po is a multiprocessing.pool.MapResult
po = p.map_async(calculate_z_serial_purepython, chunks)
# we get a list of lists back, one per chunk, so we have to
# flatten them back together
# po.get() will block until results are ready and then
# return a list of lists of results
results = po.get() # [[ints...], [ints...], []]
```

```
nagal:parallelpython_pure_python>python parallelpython_pure_python.py 1000 1000
Total elements: 250000
31250 8 31250
Starting pp with 8 local CPU workers
Submitting job with len(q) 31250, len(z) 31250
Submitting job with len(q) 31250, len(z) 31250
Submitting job with len(q) 31250, len(z) 31250
Submitting job with len(q) 31250, len(z) 31250
Submitting job with len(q) 31250, len(z) 31250
Submitting job with len(q) 31250, len(z) 31250
Submitting job with len(q) 31250, len(z) 31250
Submitting job with len(q) 31250, len(z) 31250
Job execution statistics:
 job count | % of all jobs | job time sum | time per job | job server
         8 |        100.00 |      14.2874 |     1.785928 | local
Time elapsed since server creation 3.75450515747
0 active tasks, 8 cores

None
Main took 0:00:04.008474
Total sum of elements (for validation): 1148485
```

# Parallel Python

import pp

```
# tuple of all parallel python servers to connect with
ppservers = () # use this machine
# I can't get autodiscover to work at home
#ppservers=("*",) # autodiscover on network

job_server = pp.Server(ppservers=ppservers)
# it'll autodiscover the nbr of cpus it can use if first arg not specified

print "Starting pp with", job_server.get_ncpus(), "local CPU workers"
output = []
jobs = []
for chunk in chunks:
    print "Submitting job with len(q) {}, len(z) {}".format(len(chunk[0]), len(chunk[2]))
    job = job_server.submit(calculate_z_serial_purepython, (chunk,), (), ())
    jobs.append(job)
for job in jobs:
    output_job = job()
    output += output_job
# print statistics about the run
print job_server.print_stats()
```

# Use the best algorithms and fastest tools

- Membership testing with sets and dictionaries is much faster, O(1), than searching sequences, O(n). When testing "a in b", b should be a set or dictionary instead of a list or tuple.
- String concatenation is best done with `''.join(seq)` which is an O(n) process. In contrast, using the '+' or '+=' operators can result in an O(n**2) process because new strings may be built for each intermediate step. The CPython 2.4 interpreter mitigates this issue somewhat; however, `''.join(seq)` remains the best practice.
- Many tools come in both list form and iterator form (range and xrange, map and itertools.imap, list comprehensions and generator expressions, dict.items and dict.iteritems). In general, the iterator forms are more memory friendly and more scalable. They are preferred whenever a real list is not required.
- Many core building blocks are coded in optimized C. Applications that take advantage of them can make substantial performance gains. The building blocks include all of the builtin datatypes (lists, tuples, sets, and dictionaries) and extension modules like array, itertools, and collections.deque.
- Likewise, the builtin functions run faster than hand-built equivalents. For example, map(operator.add, v1, v2) is faster than map(lambda x,y: x+y, v1, v2).
- Lists perform well as either fixed length arrays or variable length stacks. However, for queue applications using pop(0) or insert(0,v)), collections.deque() offers superior O(1) performance because it avoids the O(n) step of rebuilding a full list for each insertion or deletion.
- Custom sort ordering is best performed with Py2.4's key= option or with the traditional decorate-sort-undecorate technique. Both approaches call the key function just once per element. In contrast, sort's cmp= option is called many times per element during a sort. For example, sort(key=str.lower) is faster than sort(cmp=lambda a,b: cmp(a.lower(), b.lower())). See also TimeComplexity.

## Take advantage of interpreter optimizations

- In functions, local variables are accessed more quickly than global variables, builtins, and attribute lookups. So, it is sometimes worth localizing variable access in inner-loops. For example, the code for random.shuffle() localizes access with the line, random=self.random. That saves the shuffling loop from having to repeatedly lookup self.random. Outside of loops, the gain is minimal and rarely worth it.
- The previous recommendation is a generalization of the rule to factor constant expressions out of loops. Likewise, constant folding needs to be done manually. Inside loops, write "x=3" instead of "x=1+2".
- Function call overhead is large compared to other instructions. Accordingly, it is sometimes worth in-lining code inside time-critical loops.
- List comprehensions run a bit faster than equivalent for-loops (unless you're just going to throw away the result).
- Starting with Py2.3, the interpreter optimizes "while 1" to just a single jump. In contrast "while True" takes several more steps. While the latter is preferred for clarity, time-critical code should use the first form.
- Multiple assignment is slower than individual assignment. For example "x,y=a,b" is slower than "x=a; y=b". However, multiple assignment is faster for variable swaps. For example, "x,y=y,x" is faster than "t=x; x=y; y=t".
- Chained comparisons are faster than using the "and" operator. Write "x < y < z" instead of "x < y and y < z".
- A few fast approaches should be considered hacks and reserved for only the most demanding applications. For example, "not not x" is faster than "bool(x)".