Google Hash Code 2018

UniBG - 23 / 02 / 2018

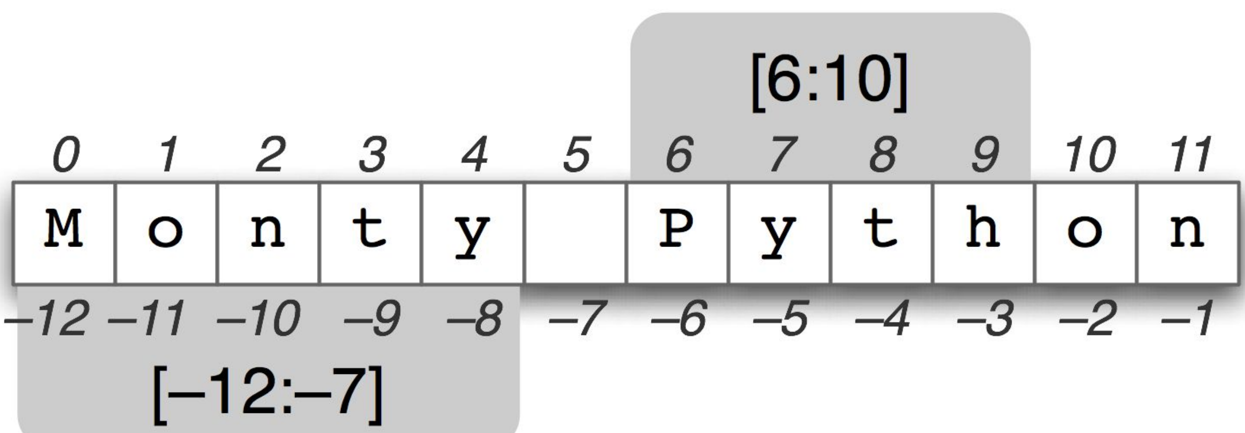
seclab.unibg.it

Python

- Dynamically typed and interpreted language
- **REPL** - Read Eval Print Loop
- Huge Standard Library
- Data structures:
  - Strings
  - Lists
  - Tuples
  - Dictionaries
  - Sets
- http://pythontutor.com/live.html
- https://learnxinyminutes.com/docs/python/
- https://learnxinyminutes.com/docs/python3/

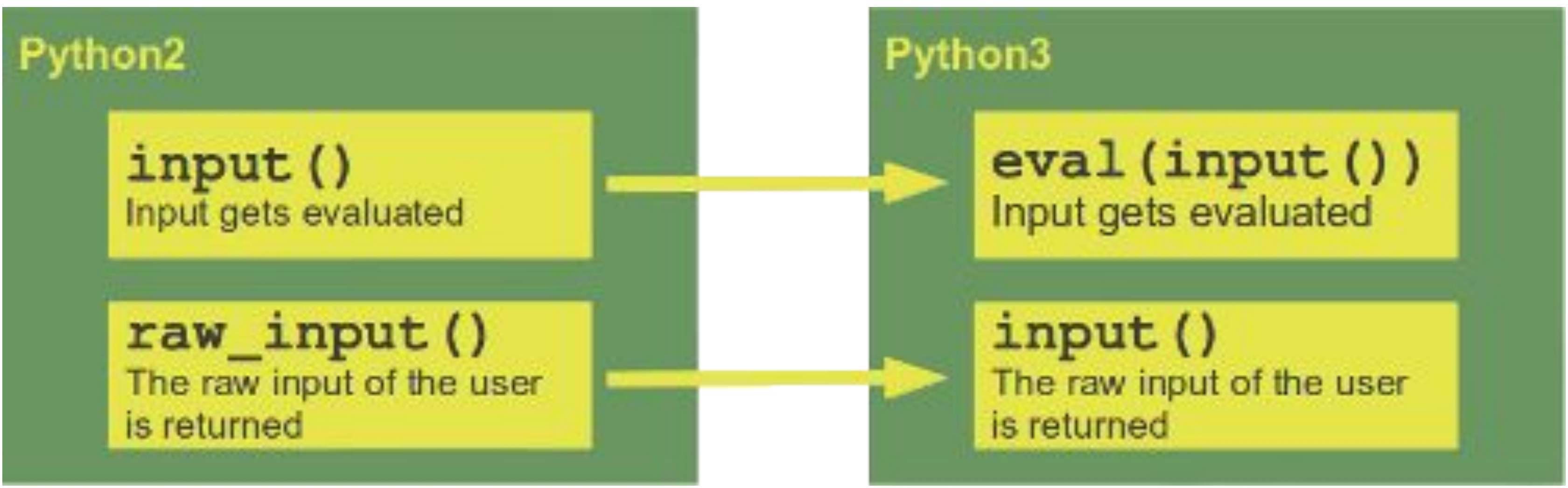
slicing

```
1.  a[start:end]        # items start through end-1
2.  a[start:]           # items start through the rest of the array
3.  a[:end]             # items from the beginning through end-1
4.  a[:]                # a copy of the whole array
5.  a[start:end:step]   # start through not past end, by step
6.  a[-1]               # last item in the array
7.  a[-2:]              # last two items in the array
8.  a[:-2]              # everything except the last two items
```



python 2 vs python 3

- Many small incompatibilities
- If you are new to python, you should use python 3



Dynamic Programming

- as seen last week, yet in Python -

The Fibonacci Sequence

1,1,2,3,5,8,13,21,34,55,89,144,233,377…

1+1=2	13+21=34
1+2=3	21+34=55
2+3=5	34+55=89
3+5=8	55+89=144
5+8=13	89+144=233
8+13=21	144+233=377

Problem:

Compute Fibonacci for N OVER 9000!

static int fibonacci_1(int n)

```
1.     static int fibonacci_1(int n) {
2.         System.out.println("computing " + n);
3.         if (n <= 2)
4.             return 1;
5.         else
6.             return fibonacci_1(n - 1) + fibonacci_1(n - 2);
7.     }
```

fibonacci_1(n)

```
1.  def fibonacci_1(n):
2.      print "Computing %d" % n
3.      if n <= 2:
4.          return 1
5.      else:
6.          return fibonacci_1(n - 1) + fibonacci_1(n - 2)
```

main

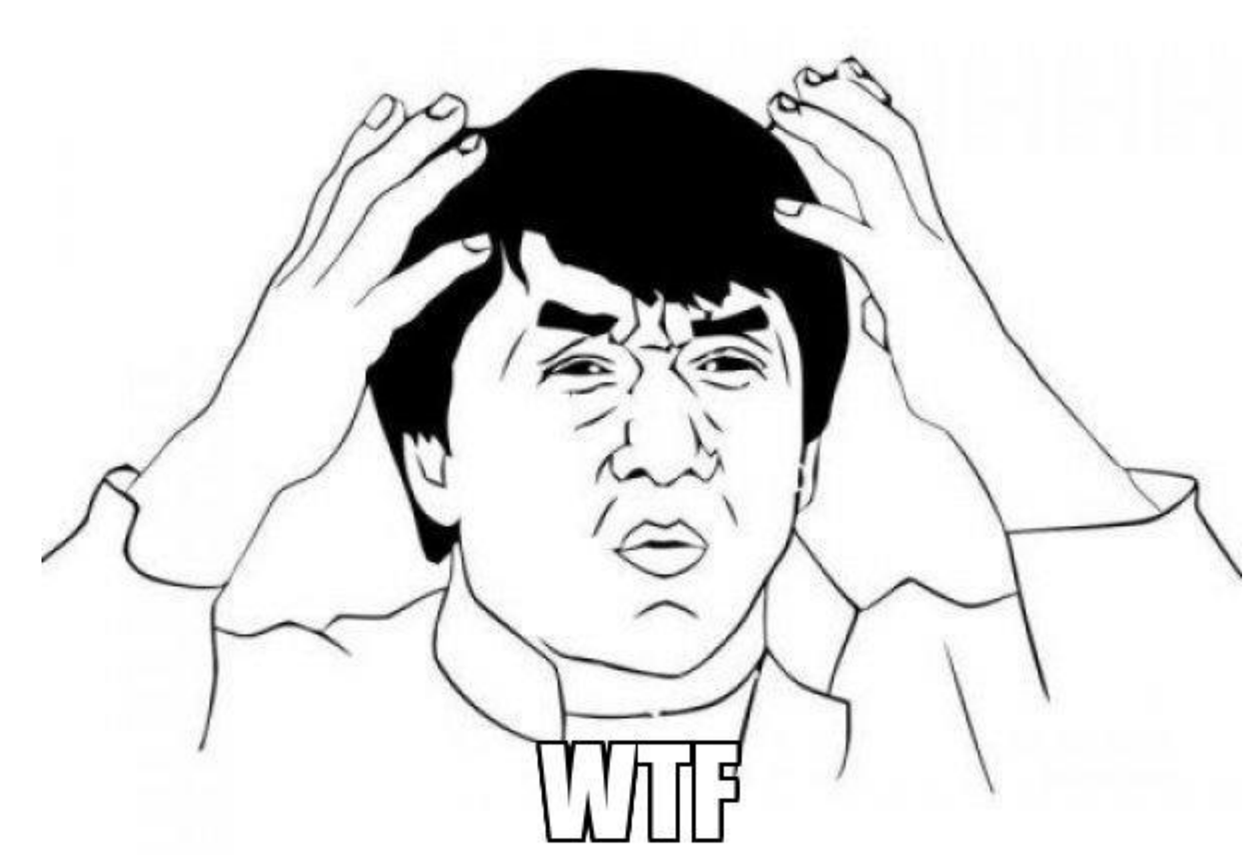
```
1.      public static void main(String[] args) {
2.          Scanner sc = new Scanner(System.in);
3.          int n = sc.nextInt();
4.          System.out.println(fibonacci_1(n));
5.      }
```

main

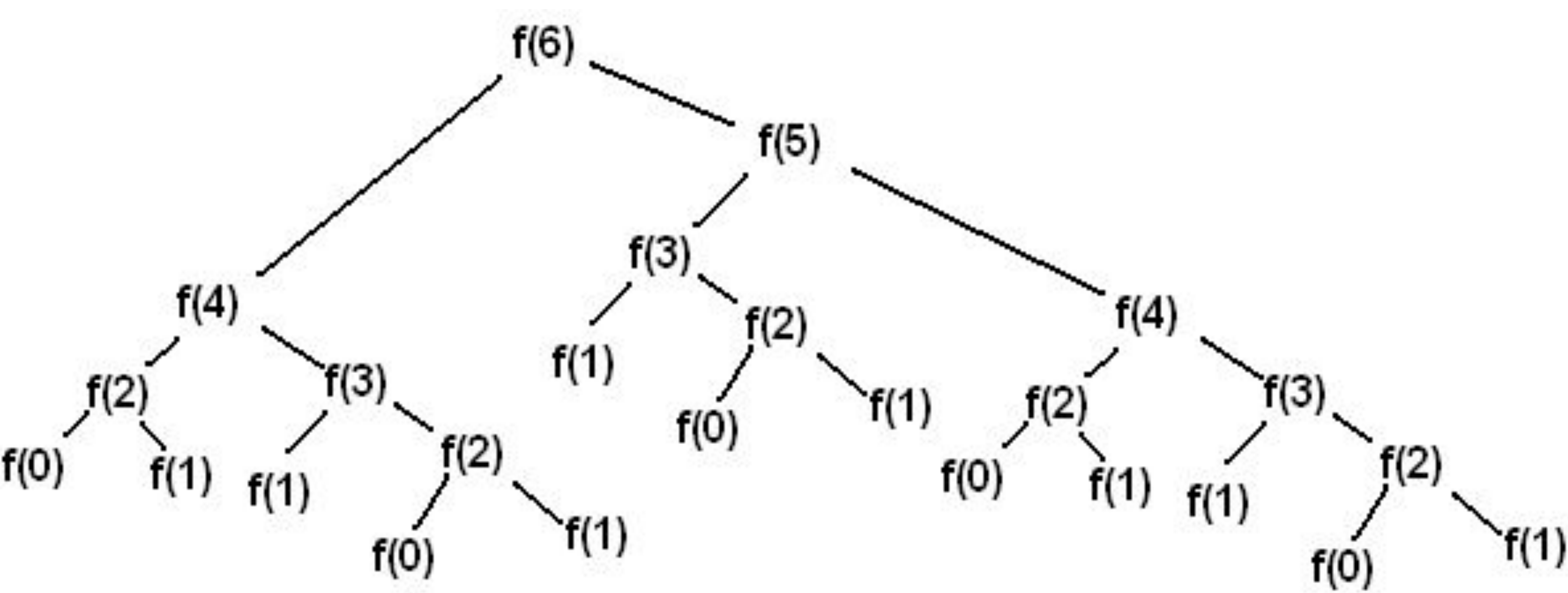
```
1.  def main():
2.      n = input("Insert number: ")
3.      print fibonacci_1(n)
4.
5.  if __name__ == '__main__':
6.      main()
```

Ok, let's try with 6

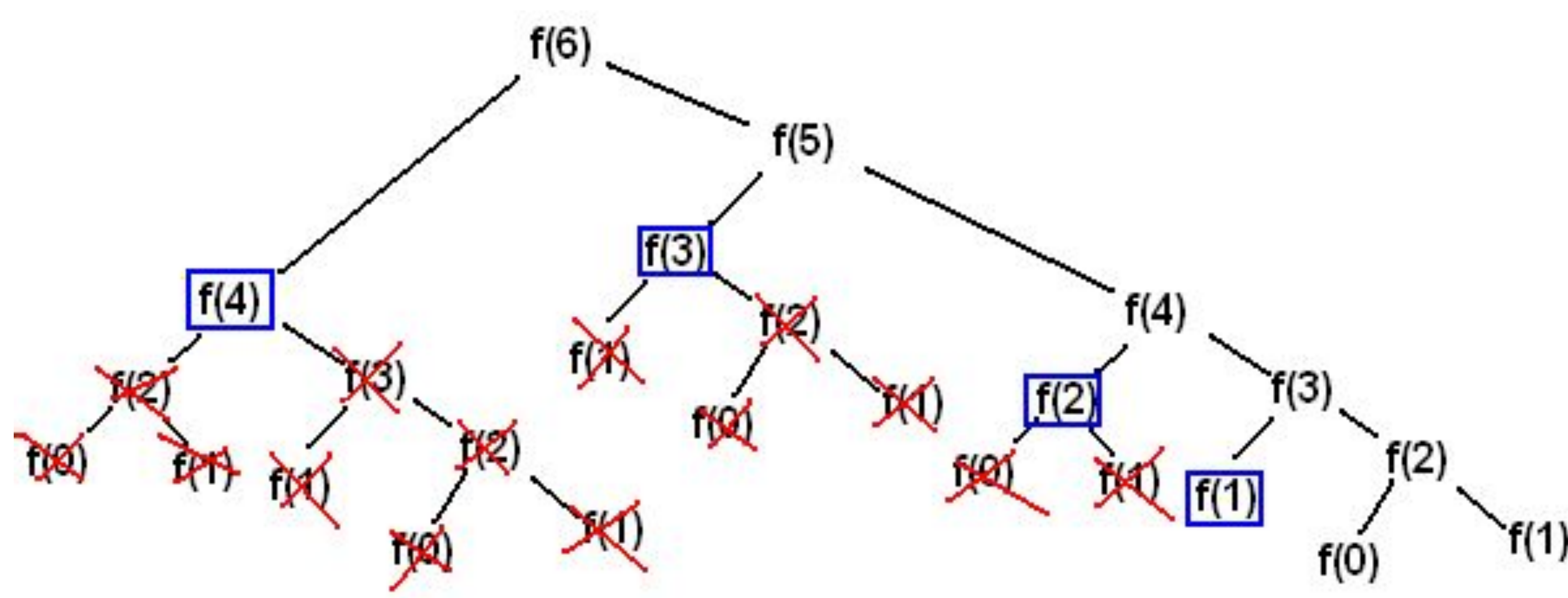
```
6
computing 6
computing 5
computing 4
computing 3
computing 2
computing 1
computing 2
computing 3
computing 2
computing 1
computing 4
computing 3
computing 2
computing 1
computing 2
8
```



Successione di Fibonacci - Stack delle chiamate



Successione di Fibonacci - Dynamic Programming



static int fibonacci_2(int n)

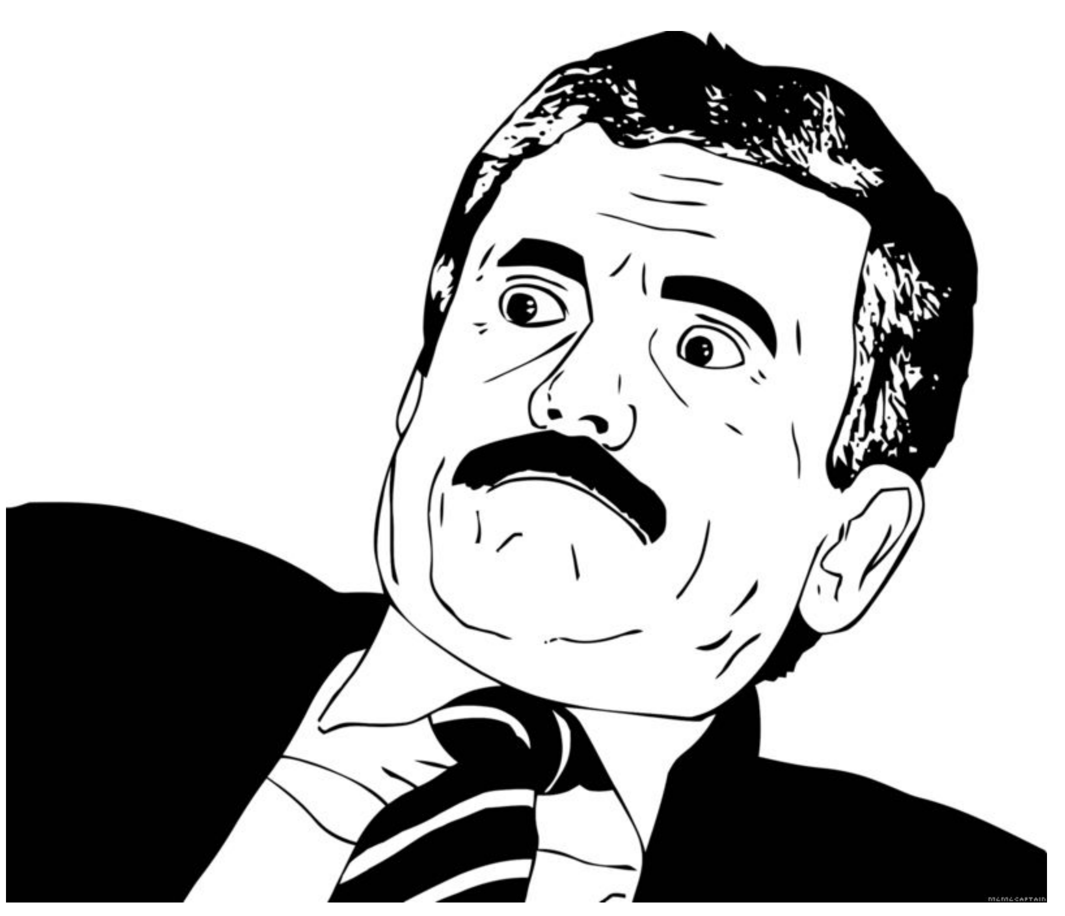
```
1.      static Map<Integer, Integer> cache_2 = new HashMap<Integer, Integer>();
2.
3.      static Integer fibonacci_2(int n) {
4.          if (cache_2.containsKey(n))
5.              return cache_2.get(n);
6.
7.          System.out.println("computing " + n);
8.          int result;
9.
10.         if (n <= 2)
11.             result = 1;
12.         else
13.             result = fibonacci_2(n - 1) + fibonacci_2(n - 2);
14.
15.         cache_2.put(n, result);
16.         return result;
17.     }
```

fibonacci_2(n)

```
1.  cache2 = {}
2.
3.  def fibonacci_2(n):
4.      if cache2.has_key(n):
5.          return cache2[n]
6.
7.      print "Computing %d" % n
8.      if n <= 2:
9.          result = 1
10.     else:
11.         result = fibonacci_2(n - 1) + fibonacci_2(n - 2)
12.
13.     cache2[n] = result
14.     return result
```

What about 100 now?

```
100
computing 100
computing 99
computing 98
computing 97
computing 96
computing 95
…
computing 5
computing 4
computing 3
computing 2
computing 1
-980107325
```



What about 100 now?

```
100
computing 100
computing 99
computing 98
computing 97
computing 96
computing 95
…
computing 5
computing 4
computing 3
computing 2
computing 1
354224848179261915075L
```

GO BIG!

```
1000
```

```
computing 1000
computing 999
computing 998
computing 997
computing 996
computing 995
…
computing 5
computing 4
computing 3
computing 2
computing 1
4346655768693745643568852767504062580256466051737178040248172908953655541794905189040387984007925516929592259308032263477520968962323987332247116164299644090653318793829896964992851600370447613779516684922 8875L
```

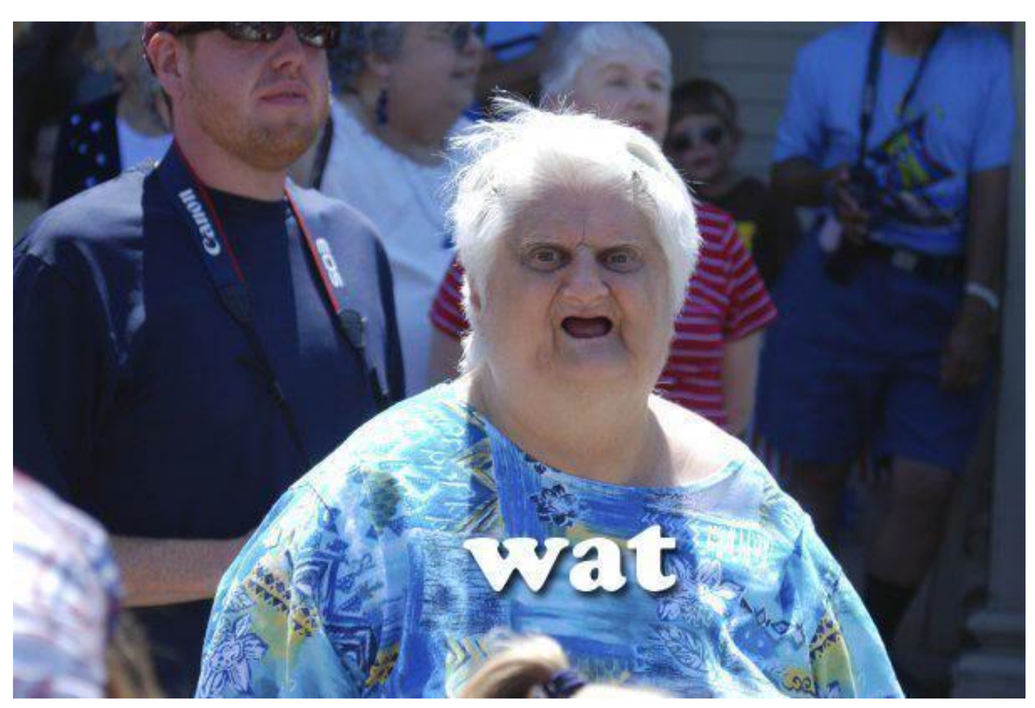
GO BIIIIIIIIIIIIIIIIIIIG!

```
10000
computing 10000
computing 9999
computing 9998
computing 9997
computing 9996
computing 9995
…
computing 4471

Exception in thread "main"
    java.lang.StackOverflowError
    …
    at com.company.Main.fibonacci_3
```

GO BIIIIIIIIIIIIIIIIIIIG!

```
10000
computing 10000
computing 9999
computing 9998
computing 9997
computing 9996
computing 9995
…
computing 9002
```




```
File "...", line xx, in fibonacci_2
   …
RuntimeError: maximum recursion depth exceeded
```

fibonacci_2(n)

```python
import sys
sys.setrecursionlimit(10000)

...

cache2 = {}

def fibonacci_2(n):
    ...
```

GO BIIIIIIIIIIIIIIIIIIIIIIIIIIIIIIIIIIIIIIIIIIIIIIIIIIIIIIIIIIIGGG!!111

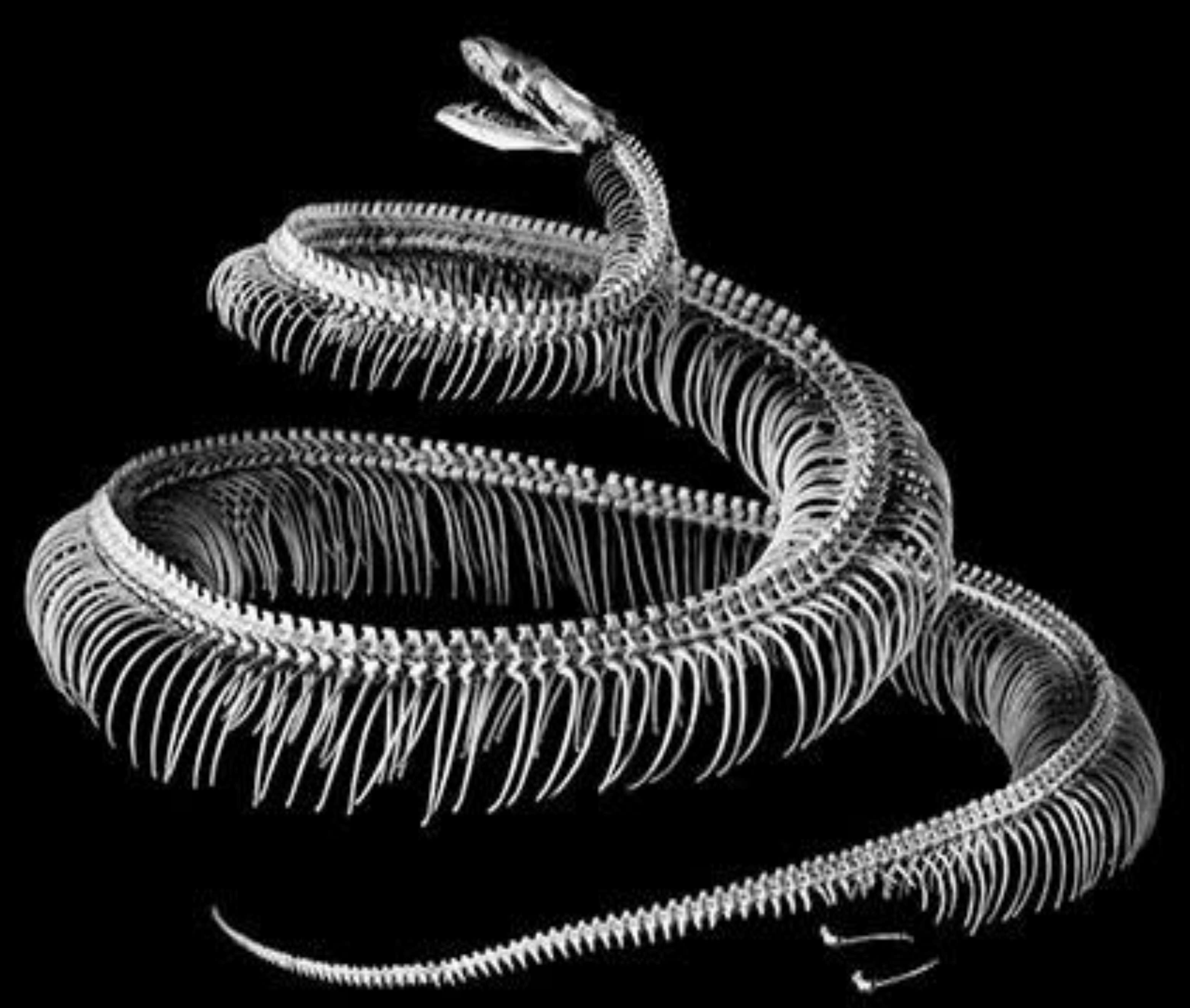
3364476487643178326662161200510754331030214846068006390656476997468008144216666236815559551
3633734025582065332680836159373734790483865268263040892463056431887354544369559827491606602
0998841839338646527313000888302692356736131351175792974378544137521305205043477016022647583
1890652789085515436615958298727968298751063120057542878345321555151038708182989697916131278 5
6265033195487140214287532698187962046936097879900350962302291026368131493195275630227837628
4415403605844025721143349611800230912082870460889239623288354615057765832712525460935911282
0392528539343462090424524892940390170623388899108584106518317336043747073790855263176432573
3993712871937587746897479926305837065742830161637408969178426378624212835258112820516370298
0893320999057079200643674262023897831114700540749984592503606335609338838319233867830561364
3535189213327973290813373264265263398976392272340788292817795358057099369104917547080893184
1056146322338217465637321248226383092103297701648054726243842374862411453093812206564914032
7510866433945175121615265453613331113140424368548051067658434935238369596534280717687753283
4823434555736671973139274627362910821067928078471803532913117677892465908993863545932789452
3777674406192240337638674004021330343297496902028328145933418826817683893072003634795623117
1031012919531697946076327375892535307725523759437884345040677155557790564504430166401194625
8097221672975861502696844314695203461493229110597067624326851599283470989128470674086200858
7135016260312071903172086094081298321581077282076353186624611278245537208532365305775956430
0725177443150515396009051686032203491632226408852488524331580515348496224348482993809050704
8348244932745373262456775587908918719080366205800959474315005240253270974699531877072437682
5907419939632265984147498193609285223945039707165443156421328157688908058783183404917434556
2705202235648464951961124602683139709750693826487066132645076650746115126775227486215986425
3071129844118262226610571635150692600298617049454250474913781151541399415506712562711971332 5
276363193960690289565028826860836224108205056243070179497617112123306607331005994736687 5L

IT'S OVER 9000!!!

aaaand it's done !

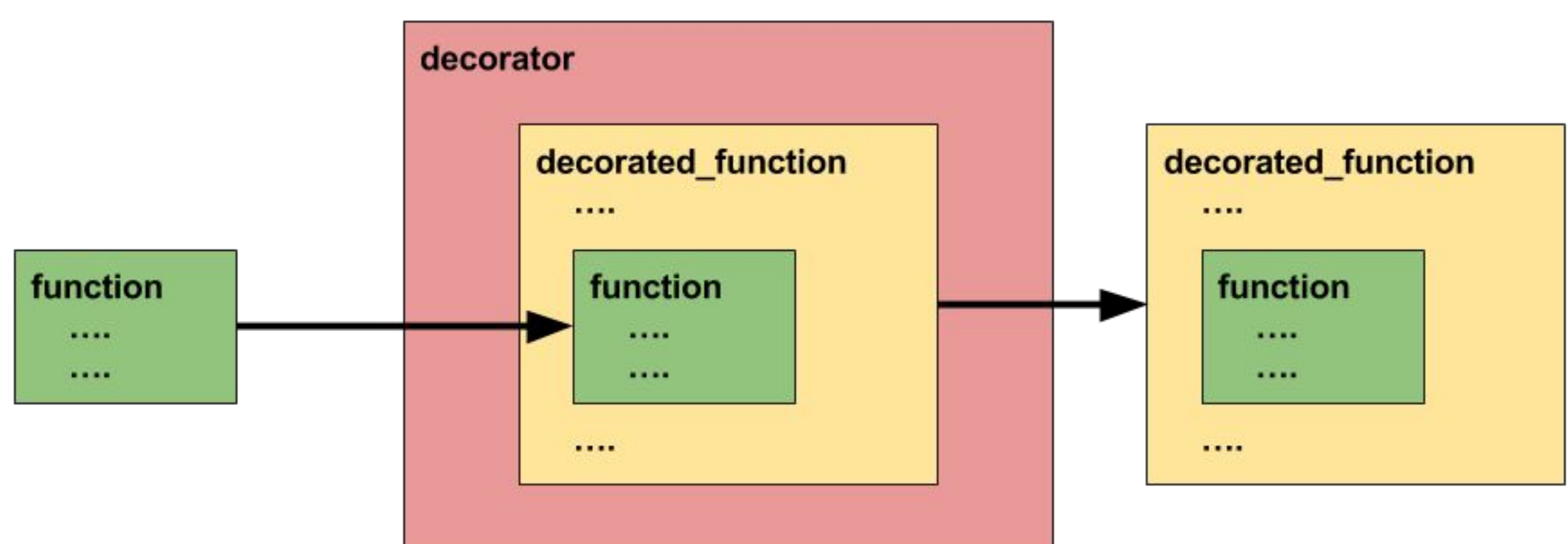
of course one should not use recursion for Fibonacci, but a loop… anyway…

```
1.  def fibonacci_5(n):
2.      a = 1
3.      b = 1
4.      for i in range(2, n):
5.          next = a + b
6.          a = b
7.          b = next
8.      return b
```



decorators

The decorator pattern is a pattern in which a function is wrapped by another function in order to add functionalities.



fibonacci_3(n)

```
1.  def memo(fn):
2.      cache = {}
3.      def _fn(n):
4.          if n not in cache:
5.              cache[n] = fn(n)
6.          return cache[n]
7.      return _fn
8.
9.  @memo
10. def fibonacci_3(n):
11.     if n <= 2:
12.         return 1
13.     return fibonacci_3(n - 1) + fibonacci_3(n - 2)
```

fibonacci_3(n)

```
1.  def memo(fn):                       # permits multiple argument functions
2.      cache = {}
3.      def _fn(*args):                 # args is a tuple of the arguments
4.          if args not in cache:
5.              cache[args] = fn(*args) # *args unpacks the arguments
6.          return cache[args]
7.      return _fn
8.
9.  @memo
10. def fibonacci_3(n):
11.     if n <= 2:
12.         return 1
13.     return fibonacci_3(n - 1) + fibonacci_3(n - 2)
```

fibonacci_3(n)

```
1.  import sys
2.  import threading
3.
4.  def main():
5.      print fibonacci_3(10000)
6.
7.  threading.stack_size(128 * 2**20) # 128MB stack
8.  sys.setrecursionlimit(2**20)  # something really big
9.
10. # only new threads get the redefined stack size
11. thread = threading.Thread(target=main)
12. thread.start()
```

pip install bigstack python-memo

```
1.  from bigstack import *
2.  from memo import *
3.
4.  @memo
5.  def fibonacci(n):
6.      print 'computing %d' % n
7.      if n <= 2: return 1
8.      return fibonacci(n-1) + fibonacci(n-2)
9.
10. @bigstack
11. def main():
12.     print fibonacci(10000)
13.
14. main()
```

DECORATORS
DECORATORS EVERYWHERE
memegenerator.net

Other Python Tips

- How can I make my python code run faster? -

lru_cache

In Python 3, the *memo* decorator is already there

@functools.**lru_cache**(*maxsize=128*, *typed=False*) ¶

Decorator to wrap a function with a memoizing callable that saves up to the *maxsize* most recent calls.

```
from functools import lru_cache as cache

@cache(maxsize=None)
def fn(x):
    ....
```

Easy parsing

```
1.  def row(fn):
2.      return map(fn, raw_input().strip().split())
3.
4.  a, b, c = row(int)
```

https://goo.gl/Qs9Gg6

```
1.  from collections import namedtuple
2.
3.  Item = namedtuple('Item', 'id value weight')
4.
5.  it = Item(1, 20, 7.5)
6.  print it.value
7.  print it.weight
```

https://goo.gl/MZqmJd

Generators - numbers

```
1.  def numbers(start=0):
2.      while True:
3.          yield start
4.          start += 1
5.
6.  for n in numbers():
7.      print n
```

tip: you can kill computations with Ctrl + C

Generators - fibonacci (again!)

```
1.  def fibonacci_generator():
2.      a = b = 1
3.      while True:
4.          yield a
5.          a, b = b, a + b
6.
7.  for x in fibonacci_generator():
8.      print x
```

tip: you can kill computations with Ctrl + C

Running with Pypy

PyPy is a fast, compliant alternative implementation of the Python language (2.7.13 and 3.5.3). It has several advantages and distinct features:

- **Speed:** thanks to its Just-in-Time compiler, Python programs often run faster on PyPy. (What is a JIT compiler?)

- **Memory usage:** memory-hungry Python programs (several hundreds of MBs or more) might end up taking less space than they do in CPython.

"If you want your code to run faster, you should probably just use PyPy."
— Guido van Rossum (creator of Python)

Using Numba

```python
from numba import jit

@jit
def f(x, y):
    # A somewhat trivial example
    return x + y
```

Similar to Pypy, but only for some functions

Useful when pypy does not work

Using Numba - nopython

```
from numba import jit

@jit(nopython=True)
def f(x, y):
    # A somewhat trivial example
    return x + y
```

Numba has two compilation modes: **nopython mode** and **object mode**. The former produces much faster code, but has limitations that can force Numba to fall back to the latter.

To prevent Numba from falling back, and instead raise an error, pass **nopython=True**.

Using Numba - nogil

```python
from numba import jit

@jit(nogil=True)
def f(x, y):
    # A somewhat trivial example
    return x + y
```

Whenever Numba optimizes Python code to native code that only works on native types and variables (rather than Python objects), it is not necessary anymore to hold Python’s global interpreter lock (GIL).

Code running with the GIL released runs concurrently with other threads executing Python or Numba code

Profiling Code

You can call it from within your code, or from the interpreter, like this:

```python
import cProfile
cProfile.run('foo()')
```

Even more usefully, you can invoke the cProfile when running a script:

```
python -m cProfile myscript.py
```

Profiling Code

```
1016 function calls in 10.000 CPU seconds


Ordered by: standard name

ncalls  tottime  percall  cumtime  percall filename:lineno(function)

    1    0.000    0.000    0.061    0.061 <string>:1(<module>)

 1000    0.051    0.000    0.051    0.000 test.py:8(foo)

   10   10.000    1.000   10.000   10.000 test.py:8(bar)

    1    0.000    0.000    0.061    0.061 {execfile}

    1    0.002    0.002    0.053    0.053 {map}

    1    0.000    0.000    0.000    0.000 {range}

    1    0.003    0.003    0.003    0.003 {sum}
```

Profiling Code

```
1016 function calls in 10.000 CPU seconds


Ordered by: standard name

ncalls  tottime  percall  cumtime  percall filename:lineno(function)

     1    0.000    0.000    0.061    0.061 <string>:1(<module>)

  1000    0.051    0.000    0.051    0.000 test.py:8(foo)

    10   10.000    1.000   10.000   10.000 test.py:8(bar)

     1    0.000    0.000    0.061    0.061 {execfile}

     1    0.002    0.002    0.053    0.053 {map}

     1    0.000    0.000    0.000    0.000 {range}

     1    0.003    0.003    0.003    0.003 {sum}
```

Profiling Code

```
1016 function calls in 10.000 CPU seconds


Ordered by: standard name
ncalls  tottime  percall  cumtime  percall filename:lineno(function)
     1    0.000    0.000    0.061    0.061 <string>:1(<module>)
  1000    0.051    0.000    0.051    0.000 test.py:8(foo)
    10   10.000    1.000   10.000   10.000 test.py:8(bar)
     1    0.000    0.000    0.061    0.061 {execfile}
     1    0.002    0.002    0.053    0.053 {map}
     1    0.000    0.000    0.000    0.000 {range}
     1    0.003    0.003    0.003    0.003 {sum}
```

Profiling Code

Other profiling / speedup suggestions here:

- https://www.ibm.com/developerworks/community/blogs/jfp/entry/Python_Meets_Julia_Micro_Performance

- https://speakerdeck.com/pyconslides/python-profiling-by-amjith-ramanujam

How to run programs

- Read from standard input (**input()**, **raw_input()**)
- Print to standard output (**print**)
- Use redirections

```
python solution.py < input.txt > output.txt
```

What about libraries?

- The first rule of PyClub is: **use pip**
- The second rule of PyClub is: **USE PIP + VENV**

Useful Python Libraries:

- **Virtualenv** github.com/pypa/virtualenv
- **Numpy** github.com/numpy/numpy
- **Pool** docs.python.org/2/library/multiprocessing

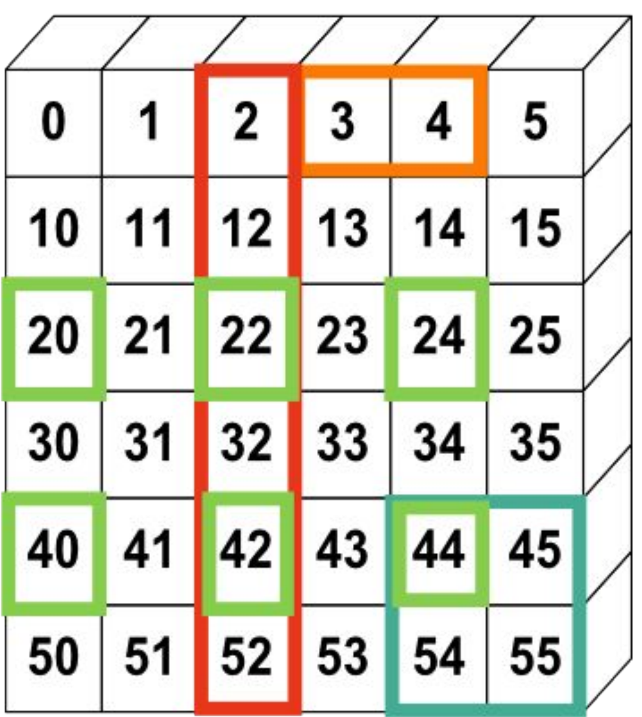
numpy slicing

```
>>> a[0,3:5]
array([3,4])

>>> a[4:,4:]
array([[44, 45],
       [54, 55]])

>>> a[:,2]
array([2,12,22,32,42,52])

>>> a[2::2,::2]
array([[20,22,24]
       [40,42,44]])
```



- Numpy is implemented in C => **super fast**
- Numpy slicing does not duplicate data => **super fast**

LET'S DO
MULTITHREADING
IN PYTHON
quickmeme.com

multiprocessing.Pool

```
1.  from multiprocessing import Pool
2.  from time import sleep
3.
4.  def slow_square(x):
5.      print 'computing square(%d)' % x
6.      sleep(2)
7.      return x * x
8.
9.
10. pool = Pool(processes=4)
11. lst = [1, 2, 3, 4, 5, 6, 7, 8, 9]
12.
13. print pool.map(slow_square, lst)
```

Greedy Algorithms

- altresì detti *golòssi* -

0/1 knapsack problem

tip: click the knapsack

0/1 Knapsack problem

```
1.  from collections import namedtuple
2.  from random import randint
3.
4.  Item = namedtuple("Item", "id weight value")
5.
6.  def build_items(n):
7.      return [Item(i, randint(1,9), randint(1,9)) for i in range(n)]
8.
9.  n = 20
10. max_weight = 15
11. items = build_items(n)
```

Bruteforce solution

```
1.  from itertools import combinations
2.
3.  def powerset(lst):
4.      for length in range(len(lst) + 1):
5.          for combination in combinations(lst, r=length):
6.              yield combination
```

Bruteforce solution

```
1.  def knapsack_bruteforce(items, max_weight):
2.      best_set = []
3.      best_value = 0
4.      for item_set in powerset(items):
5.          value = sum(item.value for item in item_set)
6.          weight = sum(item.weight for item in item_set)
7.          if weight <= max_weight and value > best_value:
8.              best_set = item_set
9.              best_value = value
10.     return best_set, best_value
11.
12. print 'bruteforce...'
13. k, v = knapsack_bruteforce(items, max_weight)
14. print 'value: %d\nknapsack: %s\n' % (v, k)
```

TOO SLOW
quickmeme.com

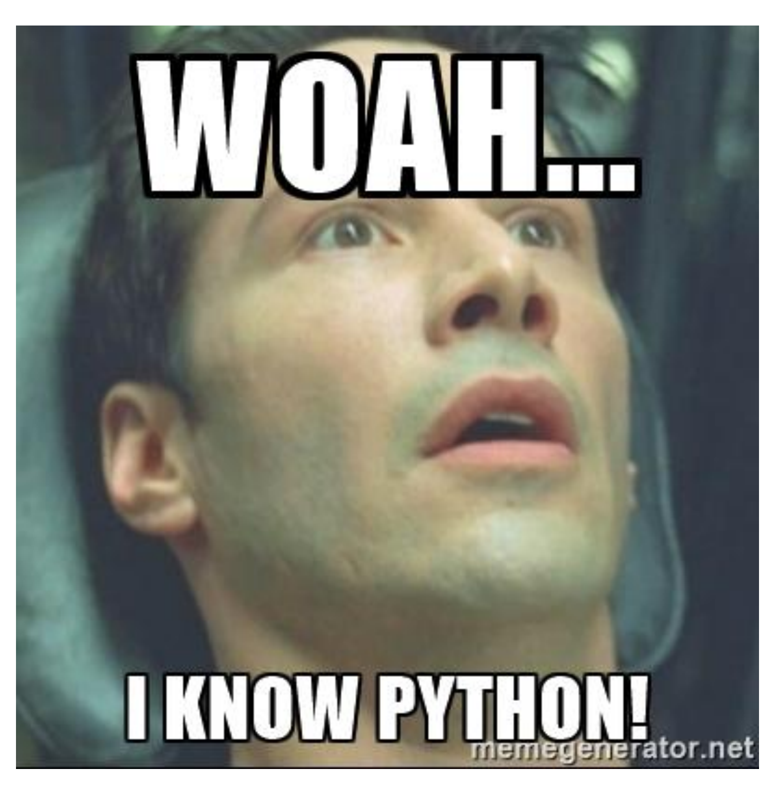
Greedy solution - 1

```
1.  def value(item): return item.value
2.
3.  def weight(item): return item.weight
4.
5.  def density(item): return float(item.value) / item.weight
6.
7.  def knapsack_greedy(items, max_weight, keyFunc):
8.      knapsack = []
9.      knapsack_value = 0
10.     remaining_weight = max_weight
11.     items = sorted(items, key=keyFunc, reverse=True)
12.
13.     for item in items:
14.         if item.weight <= remaining_weight:
15.             remaining_weight -= item.weight
16.             knapsack_value += item.value
17.             knapsack.append(item)
18.
19.     return knapsack, knapsack_value
```

Greedy solution - 2

```
1.  print 'greedy by value...'
2.  k, v = knapsack_greedy(items, max_weight, value)
3.  print 'value: %d\nknapsack: %s\n' % (v, k)
4.
5.  print 'greedy by weight...'
6.  k, v = knapsack_greedy(items, max_weight, weight)
7.  print 'value: %d\nknapsack: %s\n' % (v, k)
8.
9.  print 'greedy by density...'
10. k, v = knapsack_greedy(items, max_weight, density)
11. print 'value: %d\nknapsack: %s\n' % (v, k)
```

See the code running at:
https://repl.it/Floh/1



NOT BAD

Feedback

Ti chiediamo di dedicarci 2 minuti a compilare il form qui sotto. Nessuna risposta è obbligatoria, ma più informazioni ci darai, più ci aiuterai a fare meglio le prossime volte!

https://goo.gl/forms/jcUkKVY6f4APrmui1

seclab.unibg.it