

The Python Standard Library

...

Useful modules for quick implementations

Isn't Python alone powerful enough, you poser ?



- Tons of standard modules included in the base Python installation
- You can use them on training websites and during contests
- Many tools implementing commonly functions commonly used in competitive programming

Hold on, there's a fuckton of them, which ones should I try ?

BIG ONES

- [itertools](#) : operations on iterators, generate combinations...
- [math](#) : math
- [heapq](#) : Implementation of a priority queue
- [collections](#) : data structures
- [bisect](#) : Array bisection algorithm

Itertools (not gonna lie, it's mainly for brute-forcing)

Main goal: to create effective iterators for you to work on

Itertools is optimized for the creation of iterators

Will allow you to gain time if you know what you are looking for

Really useful to exploring basic combinatorics spaces

Itertools (you are not going to count to infinity, are you ?)

Infinite iterators:

Iterator	Arguments	Results	Example
<code>count()</code>	start, [step]	start, start+step, start+2*step, ...	<code>count(10)</code> --> 10 11 12 13 14 ...
<code>cycle()</code>	p	p0, p1, ... plast, p0, p1, ...	<code>cycle('ABCD')</code> --> A B C D A B C D ...
<code>repeat()</code>	elem [,n]	elem, elem, elem, ... endlessly or up to n times	<code>repeat(10, 3)</code> --> 10 10 10

stop counting



Itertools (who knows... it might be useful)

Iterator	Arguments	Results	Example
<code>accumulate()</code>	<code>p [,func]</code>	<code>p0, p0+p1, p0+p1+p2, ...</code>	<code>accumulate([1,2,3,4,5]) --> 1 3 6 10 15</code>
<code>compress()</code>	<code>data, selectors</code>	<code>(d[0] if s[0]), (d[1] if s[1]), ...</code>	<code>compress('ABCDEF', [1,0,1,0,1,1]) --> A C E F</code>
<code>dropwhile()</code>	<code>pred, seq</code>	<code>seq[n], seq[n+1], starting when pred fails</code>	<code>dropwhile(lambda x: x<5, [1,4,6,4,1]) --> 6 4 1</code>
<code>filterfalse()</code>	<code>pred, seq</code>	<code>elements of seq where pred(elem) is false</code>	<code>filterfalse(lambda x: x%2, range(10)) --> 0 2 4 6 8</code>
<code>takewhile()</code>	<code>pred, seq</code>	<code>seq[0], seq[1], until pred fails</code>	<code>takewhile(lambda x: x<5, [1,4,6,4,1]) --> 1 4</code>

Itertools (try doing that on $O(n^3)$)

Combinatoric iterators:

Iterator	Arguments	Results
<code>product()</code>	<code>p, q, ...</code> <code>[repeat=1]</code>	cartesian product, equivalent to a nested for-loop
<code>permutations()</code>	<code>p[, r]</code>	r-length tuples, all possible orderings, no repeated elements
<code>combinations()</code>	<code>p, r</code>	r-length tuples, in sorted order, no repeated elements
<code>combinations_with_replacement()</code>	<code>p, r</code>	r-length tuples, in sorted order, with repeated elements

Examples	Results
<code>product('ABCD', repeat=2)</code>	AA AB AC AD BA BB BC BD CA CB CC CD DA DB DC DD
<code>permutations('ABCD', 2)</code>	AB AC AD BA BC BD CA CB CD DA DB DC
<code>combinations('ABCD', 2)</code>	AB AC AD BC BD CD
<code>combinations_with_replacement('ABCD', 2)</code>	AA AB AC AD BB BC BD CC CD DD



Math, Some situational Functions...

- `math.comb(n, k)`
- `math.factorial(x)` Deprecated since version 3.9
- `math.gcd(*integers)` only two arguments were supported before 3.9
- `math.trunc(x)`
- `math.exp(x)`
- `math.log(x[, base])` `math.cos(x)`
- `math.dist(p, q)`
- `math.degrees(x)`

...and some Constants

- `math.pi`
- `math.e`
- `Math.inf` equivalent to `float('inf')`

π : 3.141592653589793

e: 2.7182818284590452

Engineers:



Heapq : a priority queue

Will be essential for Dijkstra's Algorithm $\exists(\neg \exists)$

```
from heapq import *  
  
list = [5, 2, 4, 1, 3, 2]  
heapify(list)  
  
print(heapop(list))  
heappush(list, 5)  
print(heapop(list))  
print(list)
```

Collections : data structures, containers

Deque : implementation of a 2-ended-queue

Can be a stack, as well as a queue : append left, right, push left, right

```
from collections import deque
a = deque([1, 3])
a.append(5)    # Append right
a.appendleft(2)
print(a.pop())    # Pop right
print(a.popleft())
print(a.popleft())
print(a)
```

Collections : Counter

Count occurrences in lists

Many easy exercises are based on counting => Cheese them !

```
from collections import Counter
l = [2, 2, 1, 5, 2, 5, 1, 1]
print(Counter(l))
print(Counter(l).most_common())
```

Collections : defaultdict

Dicts with elements initialized as a list, an object...

```
from collections import defaultdict
d = defaultdict(list)
d[1].append(2)
print(d.items())
```

Bisect : array binary search

Who said you have to implement your own binary search algorithm ?

Given a sorted list, get the position of an element to be inserted

```
from bisect import bisect_left

data = [6, 4, 2, 7, 2, 8, 9]
data.sort()
print(data)
print(a := bisect_left(data, 5))
data.insert(a, 5)
print(data)
```

But wait, there's more !

smol ones

- copy : hard copy an objects
- functools : magic functions
- re : Regular expressions for pattern matching
- json : you guessed it
- pprint : print, but prettier
- fractions : fraction
- base64 : base64

Copy : copy, like... for real

When you don't want to copy the pointer but the real object...

Can be useful !

```
copy.copy(x)                                # Shallow  
(pointer) copy  
copy.deepcopy(x[, memo])  # Deep (object) copy
```


Functools, cached functions

```
@lru_cache(maxsize=500) # @cache en 3.9
def factorial(n):
    return n*factorial(n-1) if n else 0
```

- Cache the result of functions
- Automatic memoization
- Results still have to be proven...

Re : regular expressions and pattern matching

Regex are dreaded, for good reasons. Though, they can be useful !

```
from re import match, findall
print(match("^[0-9]{2}$", "12"))
print(findall("[0-9]{2}", "0901023"))
```

Json : load/encode easily json objects

- Transform back and forth json files
- An easy way to store arrays/dicts

```
import json
with open("object.json", "w") as f:
    f.write(json.dumps(['foo', {'bar': ('baz', None, 1.0, 2)}]))

with open("object.json", "r") as f:
    a = f.readline()
    print(a)
    print(json.loads(a))
```

Pprint, because apes hate debuggers



```
>>> from pprint import pprint
>>> print(matrix)
[[1, 2, 3, 4, 5], [1, 2, 3, 4, 5], [1, 2, 3, 4, 5], [1, 2, 3, 4, 5], [1, 2, 3, 4, 5]]
>>> pprint(matrix)
[[1, 2, 3, 4, 5],
 [1, 2, 3, 4, 5],
 [1, 2, 3, 4, 5],
 [1, 2, 3, 4, 5],
 [1, 2, 3, 4, 5]]
```

Fractions : fractions

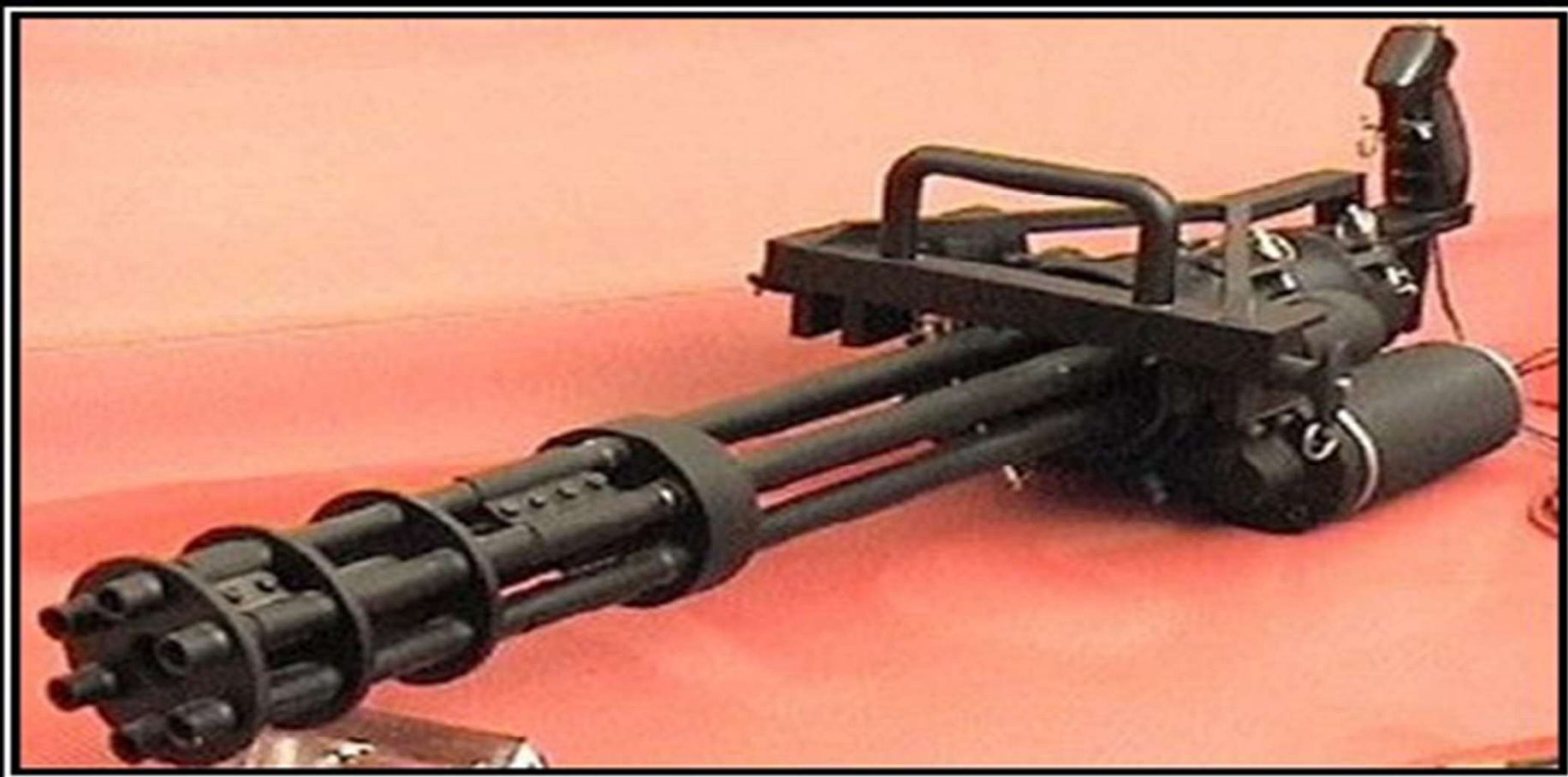
- Some problems may need to use fractions, and it's a pain in the ass to reimplement
- Once again, the PSL has a solution !

```
>>> Fraction(2,6) - Fraction(1,12)
Fraction(1, 4)
>>> Fraction(2,6) * Fraction(1,12)
Fraction(1, 36)
```

Base64 : encode and decode raw bytes

Ever seen those sequences with `=` at the end ? That's base64

```
import base64
a = base64.b64encode(b"Lalalala")
print(a)
print(base64.b64decode(a))
```



BRUTE FORCE

If it doesn't work, you're just not using enough.