20 Python Tips by Dr. DSR Murthy

1. **Use list comprehensions.**

```
cube_numbers = []
 for n in range(0,10):
  if n % 2 == 1:
    cube_numbers.append(n**3)
```

A list comprehension approach is only shorter ,faster and concise

```
cube_numbers = [n**3 for n in range(1,10) if n%2 == 1]
```

2. **Remember the built-In functions.**

Python comes with a lot of optimized and tested functions in libraries. Use it.

3. **Use xrange() instead of range().**

range() and xrange() to iterate over loops. The first (python 2.0) of these functions stored all the numbers in the range in memory and got linearly large as the range did.

xrange(), returned the generator object. When looping with this object, the numbers are in memory only on demand. Python 3 implements the xrange() functionality by default. So no xrange().

```
import sys
numbers = range(1, 1000000)
print(sys.getsizeof(numbers))
```

This returns 8000064, whereas the same range of numbers with xrange returns 40.

4**. Consider writing your own generator.**

When working with lists, consider writing own generator to take advantage of this lazy loading and memory efficiency.

Generators are particularly useful when reading a large number of large files. It's possible to process single chunks without worrying about the size of the files.

Example: web scraping and crawling recursively.

```
import requests
import re

def get_pages(link):
  pages_to_visit = []
  pages_to_visit.append(link)
  pattern = re.compile('https?')
  while pages_to_visit:
    current_page = pages_to_visit.pop(0)
    page = requests.get(current_page)
    for url in re.findall('<a href="([^"]+)">', str(page.content)):
      if url[0] == '/':
        url = current_page + url[1:]
      if pattern.match(url):
        pages_to_visit.append(url)
    yield current_page
webpage = get_pages('http://www.example.com')
for result in webpage:
  print(result)
```

This code returns a page at a time and performs an action of some sort. Without a generator, need to fetch and process at the same time or gather all the links before started processing. This code is cleaner, faster, and easier to test.

5. Use “in” if possible.

To check if membership of a list, it’s generally faster to use the “in” keyword.

```
for name in member_list:
  print('{} is a member'.format(name))
```

6**. Be lazy with module importing.**

Load the modules only when we need them. This technique helps distribute the loading time for modules more evenly, which may reduce peaks of memory usage.

7. **Use sets and unions.**

Say you wanted to get the overlapping values in two lists. You could do this using nested for loops, like this:

```
a = [1,2,3,4,5]
b = [2,3,4,5,6]
overlaps = []
for x in a:
 for y in b:
  if x==y:
   overlaps.append(x)
print(overlaps)
```

This will print the list [2, 3, 4, 5]. The number of comparisons here will get very large, very quickly.

Another approach would be:

```
a = [1,2,3,4,5]
b = [2,3,4,5,6]
overlaps = set(a) & set(b)
print(overlaps)
```

This will print the dictionary {2, 3, 4, 5}. We are on the built-in functions and getting a big speed and memory bump as a result.

8. **Remember to use multiple assignment.**

```
first_name, last_name, city = "Murthy", "Sriram", "Hyderabad"
```

we can now swap the values of variables.

x, y = y, x

This approach is much quicker and cleaner than:

```
temp = x
x = y
y = temp
```

9. **Avoid global variables.**

Using few global variables is an effective design pattern because it helps keep track of scope and unnecessary memory usage. Also, Python is faster retrieving a local variable than a global one. So, avoid that global keyword.

10. **Use join() to concatenate strings.**

concatenate strings using "+". But strings in Python are immutable, and the "+" operation involves creating a new string and copying the old content at each step.

A more efficient approach would be to use the array module to modify the individual characters and then use the join() function to re-create your final string.

```
new = "This" + "is" + "going" + "to" + "require" + "a" + "new" + "string" + "for" + "every" + "word"
print(new)
```

That code will print:

```
Thisisgoingtorequireanewstringforeveryword
```

Below code:

```
new = " ".join(["This", "will", "only", "create", "one", "string", "and", "we", "can", "add", "spaces."])
print(new)
```

will print:

```
This will only create one string and we can add spaces.This is cleaner, more elegant, and faster.
```

11. Keep up-to-date on the latest Python releases.

The Python improve performance and security in every release. So be up-to-date.

12. **Use "while 1" for an infinite loop.**

For instance, for socket , we use infite loop for listening.

So instead of using True, for faster app, use "while 1". It is only numerical comparison.

13. **Try another way.**

Think creatively and apply new coding techniques to get faster results in application.

14. Exit early.

Try to leave a function as soon as work is done reduce the indentation of program and makes it more readable. It also allows to avoid nested if statements.

```
if positive_case:
  if particular_example:
    do_something
else:
  raise exception
```

Another approach is to raise the exception early and to carry out the main action in the else part of the loop.

```
if not positive_case:
  raise exception
if not particular_example:
  raise exception
do_something
```

Now we don't need to follow the chain of logic in the conditionals and we know when this function raise an exception.

15. **Learn itertools. (gem tool)**

"gem": use the functions in itertools to create code that's fast, memory efficient, and elegant.

```
import itertools
iter = itertools.permutations(["A", "B", "C"])
list(iter)
```

This function will return all possible permutations:

```
[('A', 'B', 'C'),
 ('A', 'C', 'B'),
 ('B', 'A', 'C'),
 ('B', 'C', 'A'),
 ('C', 'A', 'B'),
 ('C', 'B', 'A')]
```

It's useful and blazingly fast!

16. **Try decorator caching.**

Memoization is a specific type of caching that optimizes software running speeds.

A cache stores the results of an operation for later use. The results could be rendered web pages or the results of complex calculations.

Fibonacci numbers: 1, 1, 2, 3, 5….

One algorithm to calculate these is:

```
def fibonacci(n):
 if n == 0: # There is no 0'th number
  return 0
 elif n == 1: # We define the first number as 1
  return 1
 return fibonacci(n - 1) + fibonacci(n-2)
```

When we use this algorithm to find the 36th Fibonacci number, fibonacci(36), the calculation took five seconds, and the answer was 14,930,352.

With caching from the standard library

```
import functools

@functools.lru_cache(maxsize=128)
def fibonacci(n):
 if n == 0:
  return 0
 elif n == 1:
  return 1
 return fibonacci(n - 1) + fibonacci(n-2)
```

In Python, a decorator function takes another function and extends its functionality.

For decorator, pass the maximum number of items to store in cache as argument.

This time the calculation took 0.7 seconds for the same answer.

17**. Use keys for sorts.**

The best way to sort items is to use keys and the default sort() method whenever possible for performance.

```
import operator
my_list = [("Jeevan", "Girish", "Singer"), ("Murthy", "sriram", "General"), ("Aravind", "laheri", "Scientist")
]
my_list.sort(key=operator.itemgetter(0))
my_list
```

This will sort the list by the first keys:

Sort by the second key, like so:

```
my_list.sort(key=operator.itemgetter(1))

my_list
```

18. Do use custom generator and iterator for extensibility

19. **Use linked lists instead of list which is array.**

The Python list datatype implements as an array. Adding an element to the start of the list is a costly operation, as every item has to be moved forward.

A linked list is a datatype in which each item has a link to the next item in the list.

An array needs the memory for the list allocated up front. That allocation can be expensive and wasteful, especially size of the array not known in advance.

A linked list allocate the memory when needed. Each item can be stored in different parts of memory, and the links join the items.

20. Use profiler tools for performance checking.