src Documentation

Release 0.1

Author

Jan 21, 2018

General

A repository collecting in one place everything I think can be useful to keep one's Pythonic zen even in the toughest job interviews, and more importantly to become a better Python Developer.

CHAPTER 1

Features

The idea is to cover:

- Definitions of fondamental concepts of Object Oriented programming and Python
- Best coding practices and examples
- Description of important packages
- Common structures and algorithms
- Common coding interview questions

CHAPTER 2

Philosophy

It is originally intended to be notes for myself but I release it in the hope it will be helpful for others. I will try my best to update it regularly and stick to the best practices.

CHAPTER 3

Testing

The code is tested with pytest. To run them:

```
cd path-to-repository/src/
pytest
```

To get more info about coverage and such:

```
pytest --cov-report term-missing --cov=. --verbose
```

Continuous Integration testing is done by Travis

CHAPTER 4

Contributing

Any help is always appreciated and if you have any suggestions or improvement, do not hesitate to create a ticket/pull request. Thank you very much for your interest in Pythonic Interviews.

CHAPTER 5

License

- Definition, tutorials and documentation licensed under CC BY-SA 4.0.
- Code source distributed under MIT license

CHAPTER 6

Definitions

Here will be defined in pythonic terms a few key aspects of Object Oriented programming.

6.1 Class

6.2 Object

CHAPTER 7

Sorting Algorithms

Module containing the most common sorting algorithms

- bubble sort
- selection sort
- insertion sort
- shell sort
- quicksort
- merge sort
- heapsort

For the record, Python built-in *sorted* uses by default timsort which is a hybrid stable sorting algorithm, running in O(n) for best case (list already sorted) to O(n log n). It does so by taking advantage of the fact that real-life lists often have some partial ordering.

src.classic_algo.sort.**bubble_sort**(*a*)
In this algo, the i-th pass starts at the first element and compare sequencially each element to the next, swapping them if necessary, up to n-i. The biggest element 'bubbles up' to the n-i position. This runs in O(n^2): n-1 passes of O(n) comparisons.

Note the use of the pythonic swap operation "a, b = b, a", not requiring the use of a temporary storage variable.

src.classic_algo.sort.**heap_sort**(*list_to_order*)

src.classic_algo.sort.**insertion_sort**(*a*)
The insertion sort uses another strategy: at the i-th pass, the i first terms are sorted and it inserts the i + 1 term where it belongs by shifting right all elements greater one notch right to create a gap to insert it. It also runs in O(n^2)

src.classic_algo.sort.**merge**(*a*, *b*)
helper function used by merge_sort combines two sorted lists into one sorted list

src.classic_algo.sort.**merge_sort**(*a*)
Based on the divide and conquer approach, this algorithm runs in O(n log n). The array is split by the middle

and each half is recursively sorted using merge_sort. The two sorted halves are then efficiently merged using the helper function above.

src.classic_algo.sort.**partition_helper**(*a*, *first*, *last*)
A left_mark index are initiated at the leftmost index available (ie not the pivot) and a right_mark at the rightmost. The left_mark is shifted right as long as a[left_mark] < pivot and the right_mark left as long as a[right_mark] > pivot. If left_mark < right_mark, the values at which the marks are stopped are swaped and the process continues until they cross. At this point, a[right_mark] and the pivot are swapped and the index of the right_mark is returned.

src.classic_algo.sort.**quick_sort**(*a*)
Another divide and conquer algorithm, quick sort relies on choosing a pivot value. The list is then partitioned using the scheme described in partition_helper. This placed the pivot in its correct position in the sorted list, the function is then called on the sublist a[:right_mark - 1] and a[right_mark+1:]

src.classic_algo.sort.**quicksort_helper**(*a*, *first*, *last*)
Helper function splitting the list at the pivot and recursively calling itself on the left and right parts of this splitpoint.

src.classic_algo.sort.**selection_sort**(*a*)
Swapping values can be an expensive operation. At the i-th pass, the selection sort finds the largest values and swap it with the value at n - i performing faster than bubble sort. Note this can be shortened same as above (not shown here for clarity)

src.classic_algo.sort.**shell_sort**(*list_to_order*)

src.classic_algo.sort.**short_bubble_sort**(*a*)
Variant of the short bubble, taking advantage of the fact we know that if no value has been swaped, the list is sorted and we can return early.

CHAPTER 8

Indices and tables

- genindex
- modindex
- search

Python Module Index

S

Index