FullStack.Cafe - Kill Your Tech Interview

Q1: What are the *built-in types* available In Python? ☆

Topics: Python

Answer:

Common *immutable* type:

1. numbers: `int()`, `float()`, `complex()`
2. immutable sequences: `str()`, `tuple()`, `frozenset()`, `bytes()`

Common *mutable* type (almost everything else):

1. mutable sequences: `list()`, `bytearray()`
2. set type: `set()`
3. mapping type: `dict()`
4. classes, class instances
5. etc.

You have to understand that Python represents all its data as objects. Some of these objects like lists and dictionaries are mutable, meaning you can change their content without changing their identity. Other objects like integers, floats, strings and tuples are objects that can not be changed.

Q2: Name some *characteristics* of Python? ☆

Topics: Python

Answer:

Here are a few key points:

- Python is an **interpreted language**. That means that, unlike languages like C and its variants, Python does not need to be compiled before it is run. Other interpreted languages include *PHP* and *Ruby*.
- Python is **dynamically typed**, this means that you don't need to state the types of variables when you declare them or anything like that. You can do things like `x=111` and then `x="I'm a string"` without error
- Python is well suited to **object orientated programming** in that it allows the definition of classes along with composition and inheritance. Python does not have access specifiers (like C++'s `public`, `private`), the justification for this point is given as "we are all adults here"
- In Python, **functions are first-class objects**. This means that they can be assigned to variables, returned from other functions and passed into functions. Classes are also first class objects
- Writing Python code is quick but running it is **often slower than compiled languages**. Fortunately, Python allows the inclusion of C based extensions so bottlenecks can be optimised away and often are. The `numpy` package is a good example of this, it's really quite quick because a lot of the number crunching it does isn't actually done by Python

Q3: How do I *modify* a string? ☆

Topics: Python

Answer:

You can't because strings are *immutable*. In most situations, you should simply construct a new string from the various parts you want to assemble it from. Work with them as lists; turn them into strings only when needed.

```
>>> s = list("Hello zorld")
>>> s
['H', 'e', 'l', 'l', 'o', ' ', 'z', 'o', 'r', 'l', 'd']
>>> s[6] = 'W'
>>> s
['H', 'e', 'l', 'l', 'o', ' ', 'W', 'o', 'r', 'l', 'd']
>>> "".join(s)
'Hello World'
```

Q4: Name some *benefits* of Python ☆☆

Topics: Python

Answer:

- Python is a **dynamic-typed** language. It means that you don't need to mention the data type of variables during their declaration.
- Python supports **object-orientated programming** as you can define classes along with the composition and inheritance.
- **Functions** in Python are like **first-class objects**. It suggests you can assign them to variables, return from other methods and pass them as arguments.
- Developing using Python is quick but running it is often slower than compiled languages.
- Python has several usages like web-based applications, test automation, data modeling, big data analytics, and much more.

Q5: What is *Lambda Functions* in Python? ☆☆

Topics: Python

Answer:

A **Lambda Function** is a small anonymous function. A lambda function can take *any* number of arguments, but can *only* have *one* expression.

Consider:

```
x = lambda a : a + 10
print(x(5)) # Output: 15
```

Q6: When to use a `tuple` vs `list` vs `dictionary` in Python? ☆☆

Topics: Python

Answer:

- Use a `tuple` to store a sequence of items that *will not change*.
- Use a `list` to store a sequence of items that *may change*.

- Use a `dictionary` when you want to associate *pairs* of two items.

Q7: What is *Negative Index* in Python? ☆☆

Topics: Python

Answer:

Negative numbers mean that you count from the right instead of the left. So, `list[-1]` refers to the last element, `list[-2]` is the second-last, and so on.

Q8: What are *local variables* and *global variables* in Python? ☆☆

Topics: Python

Answer:

- **Global Variables**: Variables declared *outside* a function or in global space are called global variables. These variables can be accessed by any function in the program.
- **Local Variables**: Any variable declared *inside* a function is known as a local variable. This variable is present in the local space and not in the global space.

Q9: What are the rules for *local* and *global* variables in Python? ☆☆

Topics: Python

Answer:

While in many or most other programming languages variables are treated as global if not declared otherwise, Python deals with variables the other way around. They are local, if not otherwise declared.

- In Python, variables that are only referenced inside a function are implicitly *global*.
- If a variable is assigned a value anywhere within the function's body, it's assumed to be a *local* unless explicitly declared as global.

Requiring global for assigned variables provides a bar against unintended side-effects.

Q10: Does Python have a *switch-case* statement? ☆☆

Topics: Python

Answer:

In Python befor 3.10, we **do not hav**e a switch-case statement. Here, you may write a switch function to use. Else, you may use a set of if-elif-else statements. To implement a function for this, we may use a dictionary.

```
def switch_demo(argument):
    switcher = {
        1: "January",
        2: "February",
        3: "March",
        4: "April",
        5: "May",
        6: "June",
```

```
        7: "July",
        8: "August",
        9: "September",
        10: "October",
        11: "November",
        12: "December"
    }
    print switcher.get(argument, "Invalid month")
```

Python 3.10 (2021) introduced the `match` - `case` statement which provides a first-class implementation of a "switch" for Python. For example:

For example:

```
def f(x):
    match x:
        case 'a':
            return 1
        case 'b':
            return 2
```

The `match` - `case` statement is considerably more powerful than this simple example.

Q11: How the *string* does get converted to a *number*? ☆☆

Topics: Python

Answer:

- To convert the string into a number the built-in functions are used like `int()` constructor. It is a data type that is used like `int ('1') == 1`.
- `float()` is also used to show the number in the format as `float('1') = 1`.
- The number by default are interpreted as decimal and if it is represented by `int('0x1')` then it gives an error as `ValueError`. In this the `int(string,base)` function takes the parameter to convert string to number in this the process will be like `int('0x1',16) == 16`. If the base parameter is defined as 0 then it is indicated by an octal and 0x indicates it as hexadecimal number.
- There is function `eval()` that can be used to convert string into number but it is a bit slower and present many security risks

Q12: What are descriptors? ☆☆

Topics: Python

Answer:

Descriptors were introduced to Python way back in version 2.2. They provide the developer with the ability to add managed attributes to objects. The methods needed to create a descriptor are `__get__`, `__set__` and `__delete__`. If you define any of these methods, then you have created a descriptor.

Descriptors power a lot of the magic of Python's internals. They are what make properties, methods and even the super function work. They are also used to implement the new style classes that were also introduced in Python 2.2.

Q13: Explain what is Linear (Sequential) Search and when may we use one? ☆☆

Topics: Searching Python JavaScript

Answer:

Linear (sequential) search goes through all possible elements in some array and compare each one with the desired element. It may take up to `O(n)` operations, where N is the size of an array and is widely considered to be horribly slow. In linear search when you perform one operation you reduce the size of the problem *by one* (when you do one operation in binary search you reduce the size of the problem *by half*). Despite it, it can still be used when:

- You need to perform this search only once,
- You are *forbidden* to rearrange the elements and you do not have any extra memory,
- The array is tiny, such as ten elements or less, or the performance is not an issue at all,
- Even though in theory other search algorithms may be faster than linear search (for instance binary search), in practice even on medium-sized arrays (around 100 items or less) it might be infeasible to use anything else. On larger arrays, it only makes sense to use other, faster search methods if the data is large enough, because the initial time to prepare (sort) the data is comparable to many linear searches,
- When the list items are arranged in order of *decreasing probability*, and these probabilities are geometrically distributed, the cost of linear search is only `O(1)`
- You have no idea what you are searching.

When you ask MySQL something like `SELECT x FROM y WHERE z = t`, and `z` is a column *without* an index, linear search is performed with all the consequences of it. This is why adding an index to *searchable* columns is important.

Complexity Analysis:

Time Complexity: O(n) **Space Complexity**: O(n)

- A linear search runs in at worst *linear time* and makes at most `n` comparisons, where `n` is the length of the list. If each element is *equally likely* to be searched, then linear search has an average case of `(n+1)/2` comparisons, but the average case can be affected if the search probabilities for each element vary.
- When the list items are arranged in order of *decreasing probability*, and these probabilities are geometrically distributed, the cost of linear search is only `O(1)`

Implementation:

JS

```
function linearSearch(array, toFind){
  for(let i = 0; i < array.length; i++){
    if(array[i] === toFind) return i;
  }
  return -1;
}
```

PY

```
# can be simply done using 'in' operator
if x in arr:
   print arr.index(x)

# If you want to implement Linear Search in Python
def search(arr, x):
    for i in range(len(arr)):
        if arr[i] == x:
            return i
```

```
    return -1
```

Q14: Why would you use the `pass` statement? ☆☆☆

Topics: Python

Answer:

Python has the syntactical requirement that code blocks *cannot be empty*. Empty code blocks are however useful in a variety of different contexts, for example, if you are designing a new class with some methods that you don't want to implement:

```
class MyClass(object):
    def meth_a(self):
        pass

    def meth_b(self):
        print "I'm meth_b"
```

If you were to leave out the pass, the code wouldn't run and you'll get an error:

```
IndentationError: expected an indented block
```

Other examples when we could use `pass`:

- Ignoring (all or) a certain type of `Exception`
- Deriving an exception class that does not add new behaviour
- Testing that code runs properly for a few test values, without caring about the results

Q15: What's the difference between `lists` and `tuples`? ☆☆☆

Topics: Python

Answer:

- The key difference is that `tuples` are immutable. This means that you cannot change the values in a tuple once you have created it.
- If you're going to need to change the values use a `List`.

Apart from tuples being immutable there is also a semantic distinction that should guide their usage. Tuples are heterogeneous data structures (i.e., their entries have different meanings), while lists are homogeneous sequences. Tuples have structure, lists have order.

One example of tuple be pairs of page and line number to reference locations in a book, e.g.:

```
my_location = (42, 11)  # page number, line number
```

You can then use this as a key in a dictionary to store notes on locations. A list on the other hand could be used to store multiple locations. Naturally one might want to add or remove locations from the list, so it makes sense

that lists are mutable. On the other hand it doesn't make sense to add or remove items from an existing location - hence tuples are immutable.

Q16: Is it possible to have *static methods* in Python? ☆☆☆

Topics: Python

Answer:

Use the `@staticmethod` decorator:

```
class MyClass(object):
    @staticmethod
    def the_static_method(x):
        print x

MyClass.the_static_method(2) # outputs 2
```

Finally, use `staticmethod` sparingly! There are very few situations where static-methods are necessary in Python, and I've seen them used many times where a separate "top-level" function would have been clearer.

Q17: Explain how does Python *memory management* work? ☆☆☆

Topics: Python

Answer:

Python - like C#, Java and many other languages -- uses *garbage collection* rather than manual memory management. You just freely create objects and the language's memory manager periodically (or when you specifically direct it to) looks for any objects that are no longer referenced by your program.

If you want to hold on to an object, just hold a reference to it. If you want the object to be freed (eventually) remove any references to it.

```
def foo(names):
  for name in names:
    print name

foo(["Eric", "Ernie", "Bert"])
foo(["Guthtrie", "Eddie", "Al"])
```

Each of these calls to foo creates a Python `list` object initialized with three values. For the duration of the foo call they are referenced by the variable names, but as soon as that function exits no variable is holding a reference to them and they are fair game for the garbage collector to delete.

Q18: What's the difference between the list methods `append()` and `extend()`? ☆☆☆

Topics: Python

Answer:

- `append` *adds* an element to a list, and

- `extend` *concatenates* the first list with another list (or another iterable, not necessarily a list).

Consider:

```
x = [1, 2, 3]
x.append([4, 5])
print (x)
# [1, 2, 3, [4, 5]]

x = [1, 2, 3]
x.extend([4, 5])
print (x)
# [1, 2, 3, 4, 5]
```

Q19: What is a Jump (or Block) Search? ☆☆☆

Topics: Searching Python JavaScript

Answer:

Jump Search **(also referred to as **Block Search) is an algorithm used to search for the position of a target element on a *sorted* data collection or structure.

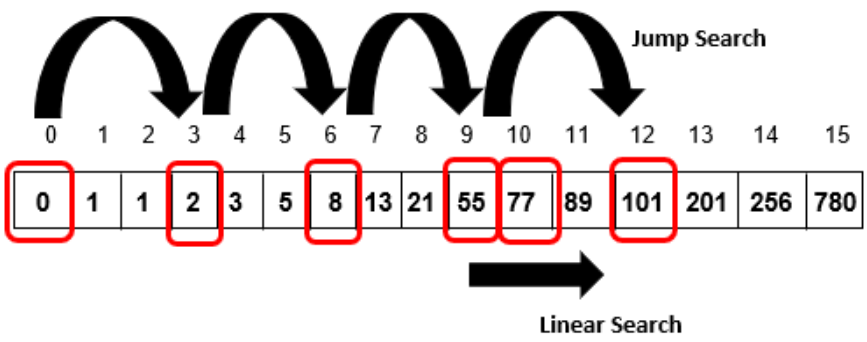
The fundamental idea behind this searching technique is to search fewer number of elements compared to linear search algorithm (which scans every element in the array to check if it matches with the element being searched or not). This can be done by skipping some fixed number of array elements or jumping ahead by fixed number of steps in every iteration.

Lets consider a sorted array `A[]` of size `n`, with indexing ranging between **0** and **n-1**, and element **x** that needs to be searched in the array `A[]`. For implementing this algorithm, a block of size **m** is also required, that can be skipped or jumped in every iteration. Thus, the algorithm works as follows:

- **Iteration 1**: if (`x==A[0]`), then success, else, if (`x > A[0]`), then jump to the next block.
- **Iteration 2**: if (`x==A[m]`), then success, else, if (`x > A[m]`), then jump to the next block.
- **Iteration 3**: if (`x==A[2m]`), then success, else, if (`x > A[2m]`), then jump to the next block.
- At any point in time, if (`x < A[km]`), then a **linear search** is performed from index `A[(k-1)m]` to `A[km]`

Consider the following inputs:

- `A[]` = {0, 1, 1, 2, 3, 5, 8, 13, 21, 55, 77, 89, 101, 201, 256, 780}
- `item` = 77



- The single most important advantage of jump search when compared to binary search is that it does not rely on the division operator (`/`).

- Jump Search is highly efficient for searching arrays especially when its only-seeks-forward behavior is favorable - its average performance makes it sit somewhere between Binary Search with its $O(log\ n)$ complexity and Linear Search with an $O(n)$ complexity.

Complexity Analysis:

Time Complexity: O(sqrt n) **Space Complexity**: O(sqrt n)

- Jump Search would consecutively jump to its very last block searching for the target element, in turn causing an `n/m` number of jumps. Additionally, if the last element's value of this block was greater than the target element, Jump Search would perform a linear search with `m-1` iterations.
- This causes Jump Search to make `(n/m)` jumps with additional `m-1` iterations. This value is minimum at $m = \sqrt{n}$. Therefore, the optimum block size is $\sqrt{n}$. Accordingly, Jump Search maintains a worst and average case efficiency of $O(\sqrt{n})$ complexity.
- The space complexity of this algorithm is $O(1)$ since it does not require any additional temporary variables or data structures for its implementation.

Implementation:

JS

```
function jumpSearch(arrayToSearch, valueToSearch){
  var length = arrayToSearch.length;
  var step = Math.floor(Math.sqrt(length));
  var index = Math.min(step, length)-1;
  var lowerBound = 0;
  while (arrayToSearch[Math.min(step, length)-1] < valueToSearch)
  {
    lowerBound = step;
    step += step;
    if (lowerBound >= length){
      return -1;
    }
  }

  var upperBound = Math.min(step, length);
  while (arrayToSearch[lowerBound] < valueToSearch)
  {
    lowerBound++;
    if (lowerBound == upperBound){
      return -1;
    }
  }
  if (arrayToSearch[lowerBound] == valueToSearch){
     return lowerBound;
  }
  return -1;
}
```

Java

```
package Searches;

public class JumpSearch implements SearchAlgorithm {

    public static void main(String[] args) {
        JumpSearch jumpSearch = new JumpSearch();
        Integer[] array = {0, 1, 2, 3, 4, 5, 6, 7, 8, 9, 10};
        for (int i = 0; i < array.length; i++) {
            assert jumpSearch.find(array, i) == i;
        }
        assert jumpSearch.find(array, -1) == -1;
        assert jumpSearch.find(array, 11) == -1;
```

```
    }

    /**
     * Jump Search algorithm implements
     *
     * @param array the array contains elements
     * @param key   to be searched
     * @return index of {@code key} if found, otherwise <tt>-1</tt>
     */
    @Override
    public <T extends Comparable<T>> int find(T[] array, T key) {
        int length = array.length; /* length of array */
        int blockSize = (int) Math.sqrt(length); /* block size to be jumped */

        int limit = blockSize;
        while (key.compareTo(array[limit]) > 0 && limit < array.length - 1) {
            limit = Math.min(limit + blockSize, array.length - 1);
        }

        for (int i = limit - blockSize; i <= limit; i++) {
            if (array[i] == key) { /* execute linear search */
                return i;
            }
        }
        return -1; /* not found */
    }
}
```

PY

```
import math
def jump_search(arr,search):
    low = 0
    interval = int(math.sqrt(len(arr)))
    for i in range(0,len(arr),interval):
        if arr[i] < search:
            low = i
        elif arr[i] == search:
            return i
        else:
            break # bigger number is found
    c=low
    for j in arr[low:]:
        if j==search:
            return c
        c+=1
    return "Not found"

arr = [ i for i in range(1,200,15)]
res = jump_search(arr, 196)
print(res)
# prints 13
```

Q20: Explain what is Interpolation Search ☆☆☆

Topics: Searching Python JavaScript

Answer:

Interpolation Search is an algorithm similar to **Binary Search** for searching for a given target value in a sorted array. It is an improvement over Binary Search for instances, where the values in a sorted array are *uniformly distributed*.

The binary search always chooses the middle of the remaining search space. On a contrast Interpolation search, at each search step, calculates (using **interpolation formula**) where in the remaining search space the target

might be present, based on the **low** and **high** values of the search space and the value of the target. For example, if the value of the key is closer to the last element, interpolation search is likely to start search toward the end side. The value actually found at this estimated position is then compared to the target value. If it is not equal, then depending on the comparison, the remaining search space is reduced to the part before or after the estimated position.

The idea of the** interpolation formula** or **partitioning logic** is to return higher value of `pos` when element to be searched is closer to `arr[hi]`. And smaller value when closer to `arr[low].`

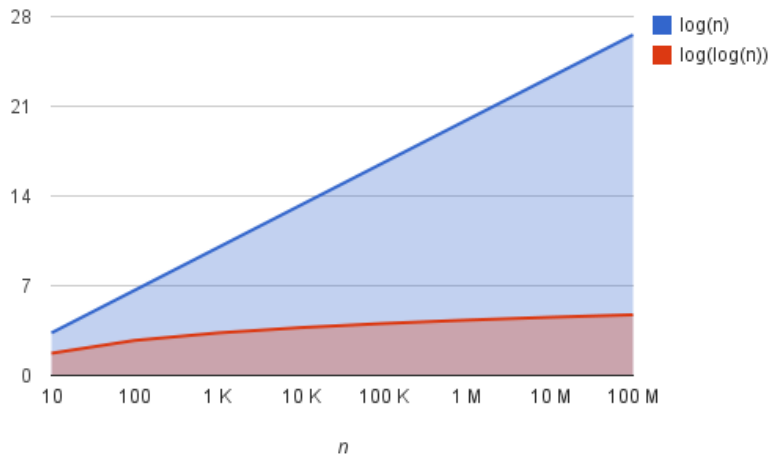
```
arr[] ==> Array where elements need to be searched
x     ==> Element to be searched
low   ==> Starting index in arr[]
hi    ==> Ending index in arr[]

pos = low + [ (x-arr[low])*(hi-low) / (arr[hi]-arr[Low]) ]
```

Complexity Analysis:

Time Complexity: O(log log n) **Space Complexity**: O(log log n)

In the interpolation search algorithm, the midpoint is computed in such a way that it gives a higher probability of obtaining our search term faster. The worst case complexity of interpolation search is `O(n)`. However, its average case complexity, under the assumption that the keys are uniformly distributed, is _`O(log log N)`.

It can be proven that Interpolation Search repeatedly reducing the problem to a subproblem of size that is the **square root of the original problem size**, and algorithm will terminate after `O(log log n)` steps (see proof here). When the values are equally dispersed into the interval this search algorithm can be extremely useful – way faster than the binary search (that divides searching space *by half*).



Implementation:

JS

```
const interpolationSearch = (array, key) => {

  // if array is empty.
  if (!array.length) {
    return -1;
  }
```

```
  let low = 0;
  let high = array.length - 1;
  while (low <= high && key >= array[low] && x <= array[high]) {

    // calculate position with
    let pos = low + Math.floor(((high - low) * (key - array[low])) / (array[high] - array[low]));

    // if all elements are same then we'll have divide by 0 or 0/0
    // which may cause NaN
    pos = Number.isNaN(pos) ? low : pos;

    if (array[pos] === key) {
      return pos;
    }

    if (array[pos] > key) {
      high = pos - 1;
    } else {
      low = pos + 1;
    }

  }

  // not found.
  return -1;
};
```

PY

```
def InterpolationSearch(data, key):
    left = 0
    right = len(data) - 1
    while left < right:
        idx = int( (right - left) * (key - data[left]) / (data[right] - data[left]) ) + left
        if data[idx] is key:
            return idx
        elif data[idx] < key:
            left = idx + 1
        else:
            right = idx - 1
    return -1
```

FullStack.Cafe - Kill Your Tech Interview

Q1: What are *Decorators* in Python? ☆☆☆

Topics: Python

Answer:

In Python, functions are the first class objects, which means that:

- Functions are objects; they can be referenced to, passed to a variable and returned from other functions as well.
- Functions can be defined inside another function and can also be passed as argument to another function.

Now:

- **Decorators** allow us to *wrap another function in order to extend the behavior* of the wrapped function, without permanently modifying it.
- **Decorators** are a very powerful and useful tool in Python since it allows programmers to *modify the behavior of function or class*.

```
@gfg_decorator
def hello_decorator():
    print("Gfg")

'''Above code is equivalent to:

def hello_decorator():
    print("Gfg")

hello_decorator = gfg_decorator(hello_decorator)'''
```

Q2: What is *Pickling* and *Unpickling*? ☆☆☆

Topics: Python

Answer:

The `pickle` module implements a fundamental, but powerful algorithm for serializing and de-serializing a Python object structure.

- **Pickling** - is the process whereby a Python object hierarchy is converted *into a byte stream*,
- **Unpickling** - is the inverse operation, whereby a byte stream is converted *back into an object hierarchy*.

```
>>> import random
>>> import pickle
>>> num_list = [random.random() for _ in range(10_000)]
>>> len(num_list)
10000
>>> num_list[:3]
[0.4877162104023087, 0.23514961430367143, 0.683895941250586]
>>> with open('nums.out', 'wb') as f:
...     pickle.dump(num_list, f)
...
```

```
>>> with open('nums.out', 'rb') as f:
...     copy_of_nums_list = pickle.load(f)
...
>>> copy_of_nums_list[:3]
[0.4877162104023087, 0.23514961430367143, 0.683895941250586]
>>> num_list == copy_of_nums_list
True
```

Q3: How can I create a *copy of an object* in Python? ☆☆☆

Topics: Python

Answer:

- To get a **fully independent copy** (deep copy) of an object you can use the `copy.deepcopy()` function. Deep copies are recursive copies of each interior object.
- For **shallow copy** use `copy.copy()`. Shallow copies are just copies of the outermost container.

Q4: What is the difference between `range` and `xrange` functions in Python? ☆☆☆

Topics: Python

Answer:

In Python 2.x:

- `range` creates a list, so if you do `range(1, 10000000)` it creates a list in memory with `9999999` elements.
- `xrange` is a generator object that evaluates lazily.

In Python 3, `range` does the equivalent of python's `xrange`, and to get the list, you have to use `list(range(...))`.

`xrange` brings you two advantages:

- You can iterate longer lists without getting a MemoryError.
- As it resolves each number lazily, if you stop iteration early, you won't waste time creating the whole list

Q5: How can you *share global variables* across modules? ☆☆☆

Topics: Python

Answer:

The canonical way to share information across modules within a single program is to **create a special configuration modu**le (often called `config` or `cfg`). Just import the configuration module in all modules of your application; the module then becomes available as a global name. Because there is only one instance of each module, any changes made to the module object get reflected everywhere.

File: `config.py`

```
x = 0 # Default value of the 'x' configuration setting
```

File: mod.py

```
import config
config.x = 1
```

File: main.py

```
import config
import mod
print config.x
```

Module variables are also often used to implement the Singleton design pattern, for the same reason.

Q6: What is *Monkey Patching* and is it ever a good idea? ☆☆☆

Topics: Python

Answer:

Monkey patching is changing the behaviour of a function or object after it has already been defined. For example:

```
import datetime
datetime.datetime.now = lambda: datetime.datetime(2012, 12, 12)
```

Most of the time it's a pretty terrible idea - it is usually best if things act in a well-defined way. One reason to monkey patch would be in **testing**. The mock package is very useful to this end.

Q7: What does this stuff mean: `*args`, `**kwargs`? Why would we use it? ☆☆☆

Topics: Python

Answer:

- Use `*args` when we *aren't sure how many* arguments are going to be passed to a function, or if we want to pass a stored list or tuple of arguments to a function.

```
>>> def print_everything(*args):
        for count, thing in enumerate(args):
...         print( '{0}. {1}'.format(count, thing))
...
>>> print_everything('apple', 'banana', 'cabbage')
0. apple
1. banana
2. cabbage
```

- `**kwargs` is used when we *don't know how many keyword arguments* will be passed to a function, or it can be used to pass the values of a dictionary as keyword arguments.

```
>>> def table_things(**kwargs):
...     for name, value in kwargs.items():
...         print( '{0} = {1}'.format(name, value))
...
>>> table_things(apple = 'fruit', cabbage = 'vegetable')
cabbage = vegetable
apple = fruit
```

Q8: Is there a tool to help find *bugs* or perform *static analysis*?

☆☆☆

Topics: Python

Answer:

- **PyChecker** is a static analysis tool that finds bugs in Python source code and warns about code complexity and style.
- **Pylint** is another tool that checks if a module satisfies a coding standard, and also makes it possible to write plug-ins to add a custom feature. In addition to the bug checking that PyChecker performs, Pylint offers some additional features such as checking line length, whether variable names are well-formed according to your coding standard, whether declared interfaces are fully implemented, and more.

Q9: Explain the `UnboundLocalError` exception and how to avoid it?

☆☆☆

Topics: Python

Problem:

Consider:

```
>>> x = 10
>>> def foo():
...     print(x)
...     x += 1
```

And the output:

```
>>> foo()
Traceback (most recent call last):
  ...
UnboundLocalError: local variable 'x' referenced before assignment
```

Why am I getting an UnboundLocalError when the variable has a value?

Solution:

When you make an assignment to a variable in a scope, that variable becomes local to that scope and shadows any similarly named variable in the outer scope. Since the last statement in foo assigns a new value to x, the compiler recognizes it as a local variable. Consequently when the earlier print(x) attempts to print the uninitialized local variable and an error results.

In the example above you can access the outer scope variable by declaring it `global`:

```
>>> x = 10
>>> def foobar():
...     global x
...     print(x)
...     x += 1
>>> foobar()
10
```

Q10: What is a `None` value? ☆☆☆

Topics: Python

Answer:

`None` is just a value that commonly is used to signify 'empty', or 'no value here'.

If you write a function, and that function doesn't use an explicit return statement, `None` is returned instead, for example.

Another example is to use `None` for default values. it is good programming practice to not use mutable objects as default values. Instead, use `None` as the default value and inside the function, check if the parameter is None and create a new list/dictionary/whatever if it is.

Consider:

```
def foo(mydict=None):
    if mydict is None:
        mydict = {}  # create a new dict for local namespace
```

Q11: What are immutable objects in Python? ☆☆☆

Topics: Python

Answer:

An object with a fixed value. Immutable objects include numbers, strings and tuples. Such an object cannot be altered. A new object has to be created if a different value has to be stored. They play an important role in places where a constant hash value is needed, for example as a key in a dictionary.

Q12: What are the key differences between Python `2` and `3`? ☆☆☆

Topics: Python

Answer:

Here are some of the key differences that a developer should be aware of:

- Text and Data instead of Unicode and 8-bit strings. Python 3.0 uses the concepts of text and (binary) data instead of Unicode strings and 8-bit strings. The biggest ramification of this is that any attempt to mix text and data in Python 3.0 raises a TypeError (to combine the two safely, you must decode bytes or encode Unicode, but you need to know the proper encoding, e.g. UTF-8)

- This addresses a longstanding pitfall for naïve Python programmers. In Python 2, mixing Unicode and 8-bit data would work if the string happened to contain only 7-bit (ASCII) bytes, but you would get UnicodeDecodeError if it contained non-ASCII values. Moreover, the exception would happen at the combination point, not at the point at which the non-ASCII characters were put into the str object. This behavior was a common source of confusion and consternation for neophyte Python programmers.

- `print` function. The print statement has been replaced with a print() function

- `xrange` – buh-bye. xrange() no longer exists (range() now behaves like xrange() used to behave, except it works with values of arbitrary size)

- API changes:

  - `zip()`, `map()` and `filter()` all now return iterators instead of lists.
  - `dict.keys()`, `dict.items()` and `dict.values()` now return 'views' instead of lists.
  - `dict.iterkeys()`, `dict.iteritems()` and `dict.itervalues()` are no longer supported.
  - Comparison operators. The ordering comparison operators (<, <=, >=, >) now raise a TypeError exception when the operands don't have a meaningful natural ordering. Some examples of the ramifications of this include:
  - Expressions like 1 < '', 0 > None or len <= len are no longer valid
  - None < None now raises a TypeError instead of returning False
  - Sorting a heterogeneous list no longer makes sense.
  - All the elements must be comparable to each other

Q13: What is the difference between `range` and `xrange`? How has this changed over time? ☆☆☆

Topics: Python

Answer:

As follows:

- `xrange` returns the xrange object while range returns the list, and uses the same memory and no matter what the range size is.
- For the most part, xrange and range are the exact same in terms of functionality. They both provide a way to generate a list of integers for you to use, however you please.
- The only difference is that range returns a Python list object and x range returns an xrange object. This means that xrange doesn't actually generate a static list at run-time like range does. It creates the values as you need them with a special technique called yielding. This technique is used with a type of object known as generators. That means that if you have a really gigantic range you'd like to generate a list for, say one billion, xrange is the function to use.
- This is especially true if you have a really memory sensitive system such as a cell phone that you are working with, as range will use as much memory as it can to create your array of integers, which can result in a Memory Error and crash your program. It's a memory hungry beast.

Q14: What is a *Callable*? ☆☆☆

Topics: Python

Answer:

- A **callable** is anything that can be called.
- A callable object allows you to use round parenthesis ( ) and eventually pass some parameters, just like functions.

Every time you define a function python creates a callable object. In the example, you could define the function **func** in these ways (it's the same):

```
class a(object):
    def __call__(self, *args):
        print 'Hello'

func = a()

# or ...
def func(*args):
    print 'Hello'
```

Q15: What is *introspection/reflection* and does Python support it? ☆☆☆

Topics: Python

Answer:

Introspection is the ability to *examine* an object at runtime. Python has a `dir()` function that supports examining the attributes of an object, `type()` to check the object type, `isinstance()`, etc.

While introspection is a passive examination of the objects, **reflection** is a more powerful tool where we could modify objects at runtime and access them dynamically. E.g.

- `setattr()` adds or modifies an object's attribute;
- `getattr()` gets the value of an attribute of an object.

It can even invoke functions dynamically:

```
getattr(my_obj, "my_func_name")()
```

Q16: What are the *Dunder/Magic/Special* methods in Python? Name a few. ☆☆☆

Topics: Python

Answer:

Dunder (derived from double underscore) methods are special/magic predefined methods in Python, with names that start and end with a double underscore. There's nothing really magical about them. Examples of these include:

- `__init__` - constructor
- `__str__`, `__repr__` - object representation (casting to string, printing)
- `__len__`, `__next__` ... - generators
- `__enter__`, `__exit__` - context managers
- `__eq__`, `__lt__`, `__gt__` - operator overloading

Q17: What are *virtualenvs*? ☆☆☆

Topics: Python

Answer:

- A **virtualenv** is what Python developers call an isolated environment for development, running, debugging Python code.
- It is used to isolate a Python interpreter together with a set of libraries and settings.
- Together with pip, it allows develop, deploy and run multiple applications on a single host, each with its own version of the Python interpreter, and a separate set of libraries.

Q18: What is the function of `self`? ☆☆☆

Topics: Python

Answer:

Self is a variable that represents *the instance of the object to itself*. In most object-oriented programming languages, this is passed to the methods as a hidden parameter that is defined by an object. But, in python, it is declared and passed explicitly. It is the first argument that gets created in the instance of the class A and the parameters to the methods are passed automatically. It refers to a separate instance of the variable for individual objects.

Let's say you have a class `ClassA` which contains a method `methodA` defined as:

```
def methodA(self, arg1, arg2): #do something
```

and `ObjectA` is an instance of this class.

Now when `ObjectA.methodA(arg1, arg2)` is called, python internally converts it for you as:

```
ClassA.methodA(ObjectA, arg1, arg2)
```

The `self` variable refers to the object itself.

Q19: What does an `x = y or z` assignment do in Python? ☆☆☆

Topics: Python

Answer:

```
x = a or b
```

If `bool(a)` returns `False`, then `x` is assigned the value of `b`.

Q20: What are the *Wheels* and *Eggs*? What is the difference? ☆☆☆

Topics: Python

Answer:

Wheel and **Egg** are both packaging formats that aim to support the use case of needing an install artifact that doesn't require building or compilation, which can be costly in testing and production workflows.

The Egg format was introduced by setuptools in 2004, whereas the Wheel format was introduced by PEP 427 in 2012.

Wheel is currently considered the standard for built and binary packaging for Python.

Here's a breakdown of the important differences between Wheel and Egg.

- Wheel has an official PEP. Egg did not.
- Wheel is a distribution format, i.e a packaging format. 1 Egg was both a distribution format and a runtime installation format (if left zipped), and was designed to be importable.
- Wheel archives do not include .pyc files. Therefore, when the distribution only contains Python files (i.e. no compiled extensions), and is compatible with Python 2 and 3, it's possible for a wheel to be "universal", similar to an sdist.
- Wheel uses PEP376-compliant .dist-info directories. Egg used .egg-info.
- Wheel has a richer file naming convention. A single wheel archive can indicate its compatibility with a number of Python language versions and implementations, ABIs, and system architectures.
- Wheel is versioned. Every wheel file contains the version of the wheel specification and the implementation that packaged it.
- Wheel is internally organized by sysconfig path type, therefore making it easier to convert to other formats.

FullStack.Cafe - Kill Your Tech Interview

Q1: What is the python `with` statement designed for? ☆☆☆

Topics: Python

Answer:

The `with` statement *simplifies exception handling* by encapsulating common preparation and cleanup tasks in so-called *context managers*.

For instance, the `open` statement is a context manager in itself, which lets you open a file, keep it open as long as the execution is in the context of the `with` statement where you used it, and close it as soon as you leave the context, no matter whether you have left it because of an exception or during regular control flow.

As a result you could do something like:

```
with open("foo.txt") as foo_file:
    data = foo_file.read()
```

OR

```
from contextlib import nested
with nested(A(), B(), C()) as(X, Y, Z):
    do_something()
```

OR (Python 3.1)

```
with open('data') as input_file, open('result', 'w') as output_file:
    for line in input_file:
        output_file.write(parse(line))
```

OR

```
lock = threading.Lock()
with lock: #Critical section of code
```

Q2: What does the Python `nonlocal` statement do (in Python 3.0 and later)? ☆☆☆

Topics: Python

Answer:

- In short, it lets you *assign values to a variable in an outer (but non-global) scope*.
- The `nonlocal` statement causes the listed identifiers to refer to previously bound variables in the nearest enclosing scope excluding globals.

For example, the counter generator can be rewritten to use this so that it looks more like the idioms of languages with closures.

```
def make_counter():
    count = 0
    def counter():
        nonlocal count
        count += 1
        return count
    return counter
```

Q3: Explain how to use *Slicing* in Python? ☆☆☆

Topics: Python

Answer:

It's pretty simple really:

```
a[start:stop]  # items start through stop-1
a[start:]      # items start through the rest of the array
a[:stop]       # items from the beginning through stop-1
a[:]           # a copy of the whole array
```

There is also the `step` value, which can be used with any of the above:

```
a[start:stop:step] # start through not past stop, by step
```

The key point to remember is that the `:stop` value represents the first value that is *not* in the selected slice. So, the difference between `stop` and `start` is the number of elements selected (if `step` is 1, the default).

The ASCII art diagram is helpful too for remembering how slices work:

```
 +---+---+---+---+---+---+
 | P | y | t | h | o | n |
 +---+---+---+---+---+---+
 0   1   2   3   4   5   6
-6  -5  -4  -3  -2  -1
```

Slicing builtin types returns a copy but that's not universal. Notably, slicing NumPy arrays returns a view that shares memory with the original.

Q4: What are *metaclasses* in Python? ☆☆☆☆

Topics: Python

Answer:

- A **metaclass** is the *class of a class*.
  - A class defines *how an instance of the class (i.e. an object) behaves* while
  - A metaclass defines *how a class behaves*.
- A class is an *instance* of a metaclass.

You can call it a 'class factory'.

Q5: What is the difference between `@staticmethod` and `@classmethod`? ☆☆☆☆

Topics: Python

Answer:

A **staticmethod** is a method that knows nothing about the class or the instance it was called on. It just gets the arguments that were passed, no implicit first argument. Its definition is immutable via inheritance.

```
class C:
    @staticmethod
    def f(arg1, arg2, ...): ...
```

A **classmethod**, on the other hand, is a method that gets passed the class it was called on, or the class of the instance it was called on, as first argument. Its definition follows Sub class, not Parent class, via inheritance.

```
class C:
   @classmethod
   def f(cls, arg1, arg2, ...): ...
```

If your method accesses other variables/methods in your class then use `@classmethod`.

Q6: What's the difference between a Python *module* and a Python *package*? ☆☆☆☆

Topics: Python

Answer:

Any Python file is a **module**, its name being the file's base name without the .py extension.

```
import my_module
```

A **package** is a collection of Python modules: while a module is a single Python file, a package is a directory of Python modules containing an additional **init**.py file, to distinguish a package from a directory that just happens to contain a bunch of Python scripts. Packages can be nested to any depth, provided that the corresponding directories contain their own **init**.py file.

Packages are modules too. They are just packaged up differently; they are formed by the combination of a directory plus **init**.py file. They are modules that can contain other modules.

```
from my_package.timing.danger.internets import function_of_love
```

Q7: What is *GIL*? ☆☆☆☆

Topics: Python

Answer:

Python has a construct called the **Global Interpreter Lock** (GIL).

The GIL makes sure that only one of your *threads* can execute at any one time. A thread acquires the GIL, does a little work, then passes the GIL onto the next thread. This happens very quickly so to the human eye it may seem like your threads are executing in parallel, but they are really just taking turns using the same CPU core. All this GIL passing adds overhead to execution.

Q8: Is it a good idea to use *multi-thread* to speed your Python code? ☆☆☆☆

Topics: Python

Answer:

Python **doesn't allow multi-threading** in the truest sense of the word. It has a multi-threading package but if you want to multi-thread to speed your code up, then it's usually not a good idea to use it.

Python has a construct called the Global Interpreter Lock (**GIL**). The GIL makes sure that only one of your 'threads' can execute at any one time. A thread acquires the GIL, does a little work, then passes the GIL onto the next thread. This happens very quickly so to the human eye it may seem like your threads are executing in parallel, but they are really just taking turns using the same CPU core. All this GIL passing adds overhead to execution.

Q9: Why are *default values* shared between objects? ☆☆☆☆

Topics: Python

Answer:

It is often expected that a function call creates new objects for *default values*. This is not what happens. Default values are created exactly once, when the function is defined. If that object is changed, like the dictionary in this example, subsequent calls to the function will refer to this changed object.

Consider:

```
def foo(mydict={}):  # Danger: shared reference to one dict for all calls
    ... compute something ...
    mydict[key] = value
    return mydict
```

The first time you call this function, mydict contains a single item. The second time, mydict contains two items because when `foo()` begins executing, mydict starts out with an item already in it.

Q10: How is *memory managed* in Python? ☆☆☆☆

Topics: Python

Answer:

Python memory is managed by Python **private heap space**.

All Python objects and data structures are located in a private heap. The programmer does not have an access to this private heap and interpreter. Like other programming language python also has garbage collector which will take care of memory management in python.Python also has an inbuilt garbage collector, which recycles all the unused memory and frees the memory, and makes it available to the heap space. The allocation of Python heap space for Python objects is done by Python memory manager. The core API gives access to some tools for the programmer to code.

Q11: Why are Python's *private* methods not actually private?

☆☆☆☆

Topics: Python

Problem:

Python gives us the ability to create `private` methods and variables within a class by prepending double underscores to the name, like this: `__myPrivateMethod()`. Can one explain this?

```
>>> class MyClass:
...     def myPublicMethod(self):
...             print 'public method'
...     def __myPrivateMethod(self):
...             print 'this is private!!'
...
>>> obj = MyClass()
>>> obj.myPublicMethod()
public method
>>> obj.__myPrivateMethod()
Traceback (most recent call last):
  File "", line 1, in
AttributeError: MyClass instance has no attribute '__myPrivateMethod'
>>> dir(obj)
['_MyClass__myPrivateMethod', '__doc__', '__module__', 'myPublicMethod']
>>> obj._MyClass__myPrivateMethod()
this is private!!
```

Solution:

In short, the **name scrambling** is used to ensure that subclasses don't accidentally override the private methods and attributes of their superclasses. *It's not designed to prevent deliberate access from outside.*

Strictly speaking, private methods are accessible outside their class, just not easily accessible. *Nothing in Python is truly private*; internally, the names of private methods and attributes are mangled and unmangled on the fly to make them seem inaccessible by their given names.

You can access the `__parse` method of the `MP3FileInfo` class by the name `_MP3FileInfo__parse`. Acknowledge that this is interesting, then promise to never, ever do it in real code. Private methods are private for a reason, but like many other things in Python, their privateness is ultimately a matter of convention, not force.

Q12: Can you explain *Closures* (as they relate to Python)? ☆☆☆☆

Topics: Python

Answer:

Objects are data with methods attached, **closures** are functions with data attached. The method of binding data to a function without actually passing them as parameters is called **closure**.

```
def make_counter():
    i = 0
    def counter(): # counter() is a closure
        nonlocal i
        i += 1
        return i
    return counter

c1 = make_counter()
c2 = make_counter()

print (c1(), c1(), c2(), c2())
# -> 1 2 1 2
```

Q13: What is the purpose of the single *underscore* `_` variable in Python? ☆☆☆☆

Topics: Python

Answer:

`_` has 4 main conventional uses in Python:

1. To hold the result of the last executed expression(/statement) in an interactive interpreter session. This precedent was set by the standard CPython interpreter, and other interpreters have followed suit
2. For translation lookup in i18n (see the gettext documentation for example), as in code like: `raise forms.ValidationError(_("Please enter a correct username"))`
3. As a general purpose "throwaway" variable name to indicate that part of a function result is being deliberately ignored (Conceptually, it is being discarded.), as in code like: `label, has_label, _ = text.partition(':')`.
4. As part of a function definition (using either `def` or `lambda`), where the signature is fixed (e.g. by a callback or parent class API), but this particular function implementation doesn't need all of the parameters, as in code like: `callback = lambda _: True`

Q14: How is `set()` implemented internally? ☆☆☆☆

Topics: Python

Problem:

I've seen people say that `set` objects in python have `O(1)` membership-checking. How are they implemented internally to allow this? What sort of data structure does it use? What other implications does that implementation have?

Solution:

- Indeed, CPython's sets are implemented as something like dictionaries with dummy values (the keys being the members of the set), with some optimization(s) that exploit this lack of values.
- So basically a **set** uses a **hashtable** as its underlying data structure. This explains the `O(1)` membership checking, since looking up an item in a hashtable is an `O(1)` operation, on average.
- Also, it worth to mention when people say sets have `O(1)` membership-checking, they are talking about the average case. In the worst case (when all hashed values collide) membership-checking is `O(n)`.

Q15: What is *MRO* in Python? How does it work? ☆☆☆☆

Topics: Python

Answer:

Method Resolution Order (MRO) it denotes the way a programming language resolves a method or attribute. Python supports classes inheriting from other classes. The class being inherited is called the Parent or Superclass, while the class that inherits is called the Child or Subclass.

In Python,** method resolution order** *defines the order in which the base classes are searched when executing a method*. First, the method or attribute is searched within a class and then it follows the order we specified while inheriting. This order is also called Linearization of a class and set of rules are called MRO (Method Resolution Order). While inheriting from another class, the interpreter needs a way to resolve the methods that are being called via an instance. Thus we need the method resolution order.

Python resolves method and attribute lookups using the C3 linearisation of the class and its parents. The C3 linearisation is **neither depth-first nor breadth-first** in complex multiple inheritance hierarchies.

Q16: What is the difference between *old style* and *new style* classes in Python? ☆☆☆☆

Topics: Python

Problem:

What is the difference between old style and new style classes in Python? When should I use one or the other?

Solution:

Declaration-wise:

New-style classes inherit from object, or from another new-style class.

```
class NewStyleClass(object):
    pass

class AnotherNewStyleClass(NewStyleClass):
    pass
```

Old-style classes don't.

```
class OldStyleClass():
    pass
```

Python 3 Note:

Python 3 doesn't support old style classes, so either form noted above results in a new-style class.

Also, **MRO (Method Resolution Order)** changed:

- Classic classes do a depth first search from left to right. Stop on first match. They do not have the **mro** attribute.
- New-style classes MRO is more complicated to synthesize in a single English sentence. One of its properties is that a Base class is only searched for once all its Derived classes have been. They have the **mro** attribute which shows the search order.

Some other notes:

- New style class objects cannot be raised unless derived from `Exception`.
- Old style classes are still marginally faster for attribute lookup.

Q17: Why Python (CPython and others) uses the *GIL*? ☆☆☆☆

Topics: Python

Answer:

In CPython, the global interpreter lock, or **GIL**, is a mutex that prevents multiple native threads from executing Python bytecodes at once. This lock is necessary mainly because *CPython's memory management is not thread-safe*.

Python has a GIL as opposed to fine-grained locking for several reasons:

- It is faster in the single-threaded case.
- It is faster in the multi-threaded case for i/o bound programs.
- It is faster in the multi-threaded case for cpu-bound programs that do their compute-intensive work in C libraries.
- It makes C extensions easier to write: there will be no switch of Python threads except where you allow it to happen (i.e. between the `Py_BEGIN_ALLOW_THREADS` and `Py_END_ALLOW_THREADS` macros).
- It makes wrapping C libraries easier. You don't have to worry about thread safety. If the library is not thread-safe, you simply keep the GIL locked while you call it.

Q18: What is an alternative to *GIL*? ☆☆☆☆

Topics: Python

Answer:

The Python interpreter is not fully thread-safe. In order to support multi-threaded Python programs, there's a global lock, called the global interpreter lock or GIL, that must be held by the current thread before it can safely access Python objects. Without the lock, even the simplest operations could cause problems in a multi-threaded program: for example, when two threads simultaneously increment the reference count of the same object, the reference count could end up being incremented only once instead of twice.

If the purpose of the GIL is to protect state from corruption, then one obvious alternative is **lock at a much finer grain (fine-grained locking)**; perhaps at a per object level. The problem with this is that although it has been demonstrated to increase the performance of multi-threaded programs, it has more overhead and single-threaded programs suffer as a result.

Q19: How to work with *transitive dependencies*? ☆☆☆☆

Topics: Python

Answer:

- **Transitive dependency** is expressing the dependency of A on C when A depends on B and B depends on C.
- If a transitive dependency is not explicitly specified in a project's `requirements.txt`, pip will grab the version of the required library specified in the project's `install_requires` section (of `setup.py`). If this section does not explicitly pin a version, you end up getting the latest version of that library.

- If your application needs a specific version of a transitive dependency, pin it yourself in your application's `requirements.txt` file. Then pip will do the right thing.

Q20: What is *Cython*? ☆☆☆☆

Topics: Python

Answer:

- **Cython** is a programming language that aims to be a superset of the Python programming language, designed to give C-like performance with code that is written mostly in Python with optional additional C-inspired syntax.
- Cython is a compiled language that is typically used to generate CPython extension modules.

FullStack.Cafe - Kill Your Tech Interview

Q1: Suppose `lst` is `[2, 33, 222, 14, 25]`. What is `lst[-1]`? ☆☆

Topics: Python

Problem:

Suppose `lst` is `[2, 33, 222, 14, 25]`, What is `lst[-1]`?

Solution:

It's `25`. Negative numbers mean that you count from the right instead of the left. So, `lst[-1]` refers to the last element, `lst[-2]` is the second-last, and so on.

Q2: Given variables `a` and `b`, switch their values so that `b` has the value of `a`, and a has the value of `b` *without using an intermediary variable* ☆☆

Topics: Python

Answer:

```
a, b = b, a
```

Q3: Return the N-th value of the Fibonacci sequence. Solve in `O(n)` time ☆☆

Topics: Fibonacci Series Data Structures Python JavaScript

Answer:

The easiest solution that comes to mind here is iteration:

```
function fib(n){
  let arr = [0, 1];
  for (let i = 2; i < n + 1; i++){
    arr.push(arr[i - 2] + arr[i -1])
  }
 return arr[n]
}
```

And output:

```
fib(4)
=> 3
```

Notice that two first numbers can not really be effectively generated by a for loop, because our loop will involve adding two numbers together, so instead of creating an empty array we assign our arr variable to `[0, 1]` that we know for a fact will always be there. After that we create a loop that starts iterating from i = 2 and adds numbers to the array until the length of the array is equal to `n + 1`. Finally, we return the number at n index of array.

Complexity Analysis:

Time Complexity: O(n) **Space Complexity**: O(n)

An algorithm in our iterative solution takes linear time to complete the task. Basically we iterate through the loop n-2 times, so Big O (notation used to describe our worst case scenario) would be simply equal to O `(n)` in this case. The space complexity is `O(n)`.

Implementation:

JS

```
function fib(n){
  let arr = [0, 1]
  for (let i = 2; i < n + 1; i++){
    arr.push(arr[i - 2] + arr[i -1])
  }
 return arr[n]
}
```

Java

```
double fibbonaci(int n){
    double prev=0d, next=1d, result=0d;
    for (int i = 0; i < n; i++) {
        result=prev+next;
        prev=next;
        next=result;
    }
    return result;
}
```

PY

```
def fib_iterative(n):
    a, b = 0, 1
    while n > 0:
        a, b = b, a + b
        n -= 1
    return a
```

Q4: Explain how Insertion Sort works ☆☆

Topics: Sorting Python JavaScript

Answer:

Insertion Sort is an *in-place*, *stable*, *comparison-based* sorting algorithm. The idea is to maintain a sub-list which is always sorted. An element which is to be 'insert'ed in this sorted sub-list, has to find its appropriate

place and then it has to be inserted there. Hence the name, **insertion sort**.

Steps on how it works:

- If it is the first element, it is already sorted.
- Pick the next element.
- Compare with all the elements in sorted sub-list.
- Shift all the the elements in sorted sub-list that is greater than the value to be sorted.
- Insert the value.
- Repeat until list is sorted.

Visualisation:

Complexity Analysis:

Time Complexity: O(n^2) **Space Complexity**: O(n^2)

- Insertion sort runs in $O(n^2)$ in its worst and average cases. It runs in `O(n)` time in its best case.
- Insertion sort performs two operations: it scans through the list, comparing each pair of elements, and it swaps elements if they are out of order. Each operation contributes to the running time of the algorithm. If the input array is already in sorted order, insertion sort compares `O(n)` elements and performs no swaps. Therefore, in the best case, insertion sort runs in `O(n)` time.
- Space complexity is `O(1)` because an extra variable key is used (as a temp variable for insertion).

Implementation:

JS

```
var insertionSort = function(a) {
    // Iterate through our array
    for (var i = 1, value; i < a.length; i++) {
        // Our array is split into two parts: values preceeding i are sorted, while others are unsorted
        // Store the unsorted value at i
        value = a[i];
        // Interate backwards through the unsorted values until we find the correct location for our
`next` value
        for (var j = i; a[j - 1] > value; j--) {
            // Shift the value to the right
            a[j] = a[j - 1];
        }
```

```
        // Once we've created an open "slot" in the correct location for our value, insert it
        a[j] = value;
    }
    // Return the sorted array
    return a;
};
```

Java

```
import java.util.Arrays;

class InsertionSort {

  void insertionSort(int array[]) {
    int size = array.length;

    for (int step = 1; step < size; step++) {
      int key = array[step];
      int j = step - 1;

      // Compare key with each element on the left of it until an element smaller than
      // it is found.
      // For descending order, change key<array[j] to key>array[j].
      while (j >= 0 && key < array[j]) {
        array[j + 1] = array[j];
        --j;
      }

      // Place key at after the element just smaller than it.
      array[j + 1] = key;
    }
  }

  // Driver code
  public static void main(String args[]) {
    int[] data = { 9, 5, 1, 4, 3 };
    InsertionSort is = new InsertionSort();
    is.insertionSort(data);
    System.out.println("Sorted Array in Ascending Order: ");
    System.out.println(Arrays.toString(data));
  }
}
```

PY

```
def insertionSort(array):

    for step in range(1, len(array)):
        key = array[step]
        j = step - 1

        # Compare key with each element on the left of it until an element smaller than it is found
        # For descending order, change key<array[j] to key>array[j].
        while j >= 0 and key < array[j]:
            array[j + 1] = array[j]
            j = j - 1

        # Place key at after the element just smaller than it.
        array[j + 1] = key

data = [9, 5, 1, 4, 3]
insertionSort(data)
print('Sorted Array in Ascending Order:')
print(data)
```

Q5: How to implement a *Tree* data-structure? Provide some code.

☆☆

Topics: Binary Tree Trees Data Structures Python

Answer:

That is a basic (generic) tree structure that can be used for `String` or any other object:

Implementation:

Java

```
public class Tree<T> {
    private Node<T> root;

    public Tree(T rootData) {
        root = new Node<T>();
        root.data = rootData;
        root.children = new ArrayList<Node<T>>();
    }

    public static class Node<T> {
        private T data;
        private Node<T> parent;
        private List<Node<T>> children;
    }
}
```

PY

Generic Tree:

```
class Tree(object):
    "Generic tree node."
    def __init__(self, name='root', children=None):
        self.name = name
        self.children = []
        if children is not None:
            for child in children:
                self.add_child(child)
    def __repr__(self):
        return self.name
    def add_child(self, node):
        assert isinstance(node, Tree)
        self.children.append(node)
#    *
#   /|\
#  1 2 +
#     / \
#    3   4
t = Tree('*', [Tree('1'),
               Tree('2'),
               Tree('+', [Tree('3'),
                          Tree('4')])])
```

Binary tree:

```
class Tree:
    def __init__(self):
```

```
        self.left = None
        self.right = None
        self.data = None
```

Q6: Convert a *Singly Linked List* to *Circular Linked List* ☆☆

Topics: Linked Lists Data Structures Python

Answer:

To convert a singly linked list to a circular linked list, we will set the next pointer of the tail node to the head pointer.

- Create a copy of the head pointer, let's say `temp`.
- Using a loop, traverse linked list till tail node (last node) using temp pointer.
- Now set the next pointer of the tail node to head node. `temp->next = head`

Implementation:

PY

```
def convertTocircular(head):
    # declare a node variable
    # start and assign head
    # node into start node.
    start = head

    # check that
    while head.next
    # not equal to null then head
    # points to next node.
    while(head.next is not None):
      head = head.next

    #
    if head.next points to null
    # then start assign to the
    # head.next node.
    head.next = start
    return start
```

Q7: What is the difference between *deep* and *shallow* copy? ☆☆☆☆

Topics: Python

Answer:

- **Shallow** copy is used when a new instance type gets created and it keeps the values that are copied in the new instance. Whereas, **deep** copy is used to store the values that are already copied.
- **Shallow** copy is used to copy the reference pointers just like it copies the values. These references point to the original objects and the changes made in any member of the class will also affect the original copy of it. Whereas, deep copy doesn't copy the reference pointers to the objects. **Deep** copy makes the reference to an object and the new object that is pointed by some other object gets stored. The changes made in the original copy won't affect any other copy that uses the object.
- **Shallow** copy allows faster execution of the program and it depends on the size of the data that is used. Whereas, **deep** copy makes it slower due to making certain copies for each object that is been called.

Q8: Why aren't Python *nested functions* called closures? ☆☆☆☆

Topics: Python

Answer:

A **closure** occurs when a function has access to a local variable from an enclosing scope that has finished its execution.

```
def make_printer(msg):
    def printer():
        print msg
    return printer

printer = make_printer('Foo!')
printer()
```

When `make_printer` is called, a new frame is put on the stack with the compiled code for the `printer` function as a constant and the value of `msg` as a local. It then creates and returns the function. Because the function `printer` references the `msg` variable, it is kept alive after the `make_printer` function has returned.

So, if your nested functions *don't*

- access variables that are local to enclosing scopes,
- do so when they are executed outside of that scope,

then *they are not closures*.

Here's an example of a nested function which is not a closure.

```
def make_printer(msg):
    def printer(msg=msg):
        print msg
    return printer

printer = make_printer("Foo!")
printer()  #Output: Foo!
```

`msg` is just a normal local variable of the function `printer` in this context.

Q9: Explain how you *reverse a generator*? ☆☆☆☆

Topics: Python

Answer:

You cannot reverse a generator in any generic way except by casting it to a sequence and creating an iterator from that. Later terms of a generator cannot necessarily be known until the earlier ones have been calculated.

Even worse, you can't know if your generator will ever hit a StopIteration exception until you hit it, so there's no way to know what there will even be a first term in your sequence.

The best you could do would be to write a `reversed_iterator` function:

```
def reversed_iterator(iter):
    return reversed(list(iter))
```

Q10: What is *Monkey Patching*? How to use it in Python? ☆☆☆☆

Topics: Python

Answer:

A **MonkeyPatch** is a piece of Python *code that extends or modifies other code at runtime* (typically at startup).

Consider:

```
from SomeOtherProduct.SomeModule import SomeClass

def speak(self):
    return "ook ook eee eee eee!"

SomeClass.speak = speak
```

It is often *used to replace a method* on the module or class level with a custom implementation.

The most common usecase is adding a workaround for a bug in a module or class when you can't replace the original code. In this case you replace the "wrong" code through monkey patching with an implementation inside your own module/package.

Q11: What are the *advantages of NumPy* over regular Python *lists*? ☆☆☆☆

Topics: Python

Answer:

NumPy's arrays are more compact than Python lists - a list of lists as you describe, in Python, would take at least 20 MB or so, while a NumPy 3D array with single-precision floats in the cells would fit in 4 MB. Access to reading and writing items is also faster with NumPy.

The difference is mostly due to "indirectness" - a Python list is an array of pointers to Python objects, at least 4 bytes per pointer plus 16 bytes for even the smallest Python object (4 for type pointer, 4 for reference count, 4 for value - and the memory allocators rounds up to 16). A NumPy array is an array of uniform values -- single-precision numbers take 4 bytes each, double-precision ones, 8 bytes. Less flexible, but you pay substantially for the flexibility of standard Python lists.

NumPy is not just more efficient; it is also more convenient. You get a lot of vector and matrix operations for free, which sometimes allows one to avoid unnecessary work. And they are also efficiently implemented.

Q12: What is the difference between a function decorated with `@staticmethod` and one decorated with `@classmethod`? ☆☆☆☆

Topics: Python

Answer:

- A **staticmethod** is a method that knows nothing about the class or instance it was called on. It just gets the arguments that were passed, no implicit first argument. it's a way of putting a function into a class (because it logically belongs there), while indicating that it does not require access to the class. You can aslo just use a module function instead of a staticmethod.

- A **classmethod**, on the other hand, is a method that gets passed the class it was called on, or the class of the instance it was called on, as first argument. This is useful when you want the method to be a factory for the class: since it gets the actual class it was called on as first argument, you can always instantiate the right class, even when subclasses are involved. Observe for instance how `dict.fromkeys()`, a classmethod, returns an instance of the subclass when called on a subclass:

```
>>> class DictSubclass(dict):
...     def __repr__(self):
...         return "DictSubclass"
...
>>> dict.fromkeys("abc")
{'a': None, 'c': None, 'b': None}
>>> DictSubclass.fromkeys("abc")
DictSubclass
>>>
```

Q13: Why would you use *metaclasses*? ☆☆☆☆☆

Topics: Python

Answer:

Well, usually you don't. The main use case for a metaclass is creating an API. A typical example of this is the Django ORM.

It allows you to define something like this:

```
class Person(models.Model):
    name = models.CharField(max_length=30)
    age = models.IntegerField()
```

And if you do this:

```
guy = Person(name='bob', age='35')
print(guy.age)
```

It won't return an `IntegerField` object. It will return an `int`, and can even take it directly from the database.

This is possible because `models.Model` defines `__metaclass__` and it uses some magic that will turn the Person you just defined with simple statements into a complex hook to a database field.

Django makes something complex look simple by exposing a simple API and using **metaclasses**, recreating code from this API to do the real job behind the scenes.

Q14: Describe Python's *Garbage Collection mechanism* in brief ☆☆☆☆☆

Topics: Python

Answer:

A lot can be said here. There are a few main points that you should mention:

- Python maintains a count of the number of references to each object in memory. If a reference count goes to zero then the associated object is no longer live and the memory allocated to that object can be freed up for something else
- occasionally things called "reference cycles" happen. The garbage collector periodically looks for these and cleans them up. An example would be if you have two objects `o1` and `o2` such that `o1.x == o2` and `o2.x == o1`. If `o1` and `o2` are not referenced by anything else then they shouldn't be live. But each of them has a reference count of 1.
- Certain heuristics are used to speed up garbage collection. For example, recently created objects are more likely to be dead. As objects are created, the garbage collector assigns them to generations. Each object gets one generation, and younger generations are dealt with first.

This explanation is CPython specific.

Q15: Is there a simple, elegant way to *define singletons*? ☆☆☆☆☆

Topics: Python

Answer:

Use a **metaclass**:

```
class Singleton(type):
    _instances = {}
    def __call__(cls, *args, **kwargs):
        if cls not in cls._instances:
            cls._instances[cls] = super(Singleton, cls).__call__(*args, **kwargs)
        return cls._instances[cls]

#Python2
class MyClass(BaseClass):
    __metaclass__ = Singleton

#Python3
class MyClass(BaseClass, metaclass=Singleton):
    pass
```

In general, it **makes sense** to use a metaclass to implement a singleton. A singleton is special because is created only once, and a metaclass is the way you customize the creation of a class. Using a metaclass gives you more control in case you need to customize the singleton class definitions in other ways.

Another approach is to use **Modules**:

```
class Foo(object):
     pass

some_global_variable = Foo()
```

Modules are imported only once, everything else is overthinking. Don't use singletons and try not to use globals.

Q16: What does Python optimisation (`-O` or `PYTHONOPTIMIZE`) do? ☆☆☆☆☆

Topics: Python

Answer:

In Python 2.7, `-O` has the following effect:

- the byte code extension changes to `.pyo`
- `sys.flags.optimize` gets set to
- `__debug__` is False
- asserts don't get executed

In addition `-OO` has the following effect:

- `sys.flags.optimize` gets set to 2
- doc strings are not available

Q17: Is there any downside to the `-O` flag apart from missing on the built-in debugging information? ☆☆☆☆☆

Topics: Python

Answer:

Many python modules that assume `docstrings` are available, and would break if that optimization level is used, for instance at the company where I work, raw sql is placed in `docstrings`, and executed by way of function decorators (not even kidding).

Somewhat less frequently, `assert` is used to perform logic functions, rather than merely declare the invariant expectations of a point in code, and so any code like that would also break.

Q18: Why use `else` in `try/except` construct in Python? ☆☆☆☆☆

Topics: Python

Problem:

Why have the code that must be executed if the try clause does not raise an exception within the try construct? Why not simply have it follow the try/except at the same indentation level?

Solution:

The `else` block is only executed if the code in the `try` doesn't raise an exception; if you put the code outside of the `else` block, it'd happen regardless of exceptions. Also, it happens before the `finally`, which is generally important.

This is generally useful when you have a brief setup or verification section that may error, followed by a block where you use the resources you set up in which you don't want to hide errors. You can't put the code in the `try` because errors may go to `except` clauses when you want them to propagate. You can't put it outside of the construct, because the resources definitely aren't available there, either because setup failed or because the `finally` tore everything down. Thus, you have an `else` block.

Consider:

```
lis = range(100)
ind = 50
try:
    lis[ind]
except:
    pass
else:
    #Run this statement only if the exception was not raised
    print "The index was okay:",ind

ind = 101

try:
    lis[ind]
except:
    pass
print "The index was okay:",ind  # this gets executes regardless of the exception
```

Q19: Why isn't all *memory freed* when Python *exits*? ☆☆☆☆☆

Topics: Python

Answer:

Objects referenced from the global namespaces of Python modules are not always deallocated when Python exits. This may happen if there are circular references. There are also certain bits of memory that are allocated by the C library that are impossible to free (e.g. a tool like Purify will complain about these). Python is, however, aggressive about cleaning up memory on exit and does try to destroy every single object.

If you want to force Python to delete certain things on deallocation, you can use the **atexit** module to register one or more exit functions to handle those deletions.

Q20: What is a global interpreter lock (*GIL*) and why is it an issue? ☆☆☆☆☆

Topics: Python

Problem:

A lot of noise has been made around removing the GIL from Python, and I'd like to understand why that is so important.

Solution:

There are several implementations of Python, for example, CPython, IronPython, RPython, etc.

In CPython, the global interpreter lock, or GIL, is a mutex that prevents multiple native threads from executing Python bytecodes at once. This lock is necessary mainly because CPython's memory management is not thread-safe.

- The **GIL is controversial** because it prevents multithreaded CPython programs from taking full advantage of multiprocessor systems in certain situations. Note that potentially blocking or long-running operations, such as I/O, image processing, and NumPy number crunching, happen outside the GIL. Therefore it is only in multithreaded programs that spend a lot of time inside the GIL, interpreting CPython bytecode, that the GIL becomes a bottleneck.

- The **GIL is a problem if**, and only if, you are doing CPU-intensive work in pure Python. Here you can get cleaner design using processes and message-passing (e.g. mpi4py).

FullStack.Cafe - Kill Your Tech Interview

Q1: How to make a *flat list out of list of lists*? ☆☆☆

Topics: Python

Answer:

Given a list of lists `l` consider:

```
flat_list = []
for sublist in l:
    for item in sublist:
        flat_list.append(item)
```

Or using *lambda function*:

```
flatten = lambda l: [item for sublist in l for item in sublist]
print(flatten([[1],[2],[3],[4,5]]))
# Output: [1, 2, 3, 4, 5]
```

Q2: How do I check if *a list is empty*? ☆☆☆

Topics: Python

Answer:

Based on PEP 8 style guide, for sequences (strings, lists, tuples), use the fact that empty sequences are *false*:

```
if not list:
  print("List is empty")
```

My problem with `if not list: ...` is that it gives the false impression that li is a boolean variable.

So more clear, explicit way would be:

```
if len(list) == 0:
    print('the list is empty')
```

Q3: What is the most efficient way to *concatenate* many strings together? ☆☆☆

Topics: Python

Answer:

`str` and `bytes` objects are immutable, therefore concatenating many strings together is inefficient as each concatenation creates a new object. In the general case, the total runtime cost is quadratic in the total string length.

To accumulate many `str` objects, I would recommend to place them into a list and call `str.join()` at the end:

```
chunks = []
for s in my_strings:
    chunks.append(s)
result = ''.join(chunks)
```

Q4: Is this *valid* in Python and why? ☆☆☆

Topics: Python

Problem:

Consider:

```
def my_function():
  print my_function.what
my_function.what = "right?"
my_function() # Prints "right?"
```

Solution:

It is **valid**. In Python, everything is an *object*, including *functions*. But if we don't have `what` defined, we will get an `Attribute error`.

Q5: Write a program to check whether the object is of *a class or its subclass* ☆☆☆

Topics: Python

Answer:

There is a method which is built-in to show the instances of an object that consists of many classes by providing a tuple in a table instead of individual classes. The method is `isinstance(obj,cls)`

`isinstance(obj, (class1, class2, ...))` is used to check the object's presence in one of the classes. The built in types can also have many formats of the same function like `isinstance(obj, str)` or `isinstance(obj, (int, long, float, complex))`:

```
def search(obj):
  if isinstance(obj, box):
  # This is the code that is given for the box and write the program in the object
  elif isinstance(obj, Document):
  # This is the code that searches the document and writes the values in it
  elif

obj.search()
#This is the function used to search the object's class.
```

Q6: After executing the above code, what is the value of `y`? ☆☆☆

Topics: Python

Problem:

```
>>> x = 100
>>> y = x
>>> x = 200
```

After executing the above code, what is the value of `y`?

Solution:

```
>>> print(y)
100
```

Assignment in Python means one thing, and one thing only: **The variable named on the left should now refer to the value on the right.**

In other words, when I said:

```
y = x
```

Python doesn't read this as, "y should now refer to the variable x." Rather, it read it as, "y should now refer to whatever value x refers to."

Because `x` refers to the integer 100, `y` now refers to the integer 100. After these two assignments ("x = 100" and "y = x"), there are now two references to the integer 100 that didn't previously exist.

When we say that "x = 200", we're removing one of those references, such that `x` no longer refers to 100. Instead, `x` will now refer to the integer 200.

Q7: Floyd's Cycle Detect Algorithm: How to detect a Cycle (or Loop) in a Linked List? ☆☆☆

Topics: Linked Lists Python

Answer:

You can make use of **Floyd's cycle-finding algorithm**, also known as *tortoise and hare algorithm*.

The idea is to have two references to the list and move them at **different speeds**. Move one forward by `1` node and the other by `2` nodes.

- If the linked list has a loop they will *definitely* meet.
- Else either of the two references(or their `next`) will become `null`.

Complexity Analysis:

Time Complexity: O(n) **Space Complexity**: O(n)

We only use two nodes (slow and fast) so the space complexity is `O(1)`.

Implementation:

Java

```
boolean hasLoop(Node first) {
    Node slow = first;
    Node fast = first;

    while (fast != null && fast.next != null) {
        slow = slow.next;          // 1 hop
        fast = fast.next.next;     // 2 hops

        if (slow == fast)  // fast caught up to slow, so there is a loop
            return true;
    }
    return false;  // fast reached null, so the list terminates
}
```

PY

```
def has_cycle(head):
    if head == None:
        return False

    tortoise = hare = head

    while(tortoise or hare or hare.next):

        if hare.next == None:
            return False

        if tortoise == hare.next or tortoise == hare.next.next:
            return True

        tortoise = tortoise.next
        hare = hare.next.next

    return False
```

Q8: Display `startNumber` to `endNumber` only from Fibonacci Sequence ☆☆☆

Topics: Fibonacci Series Python

Answer:

If your language supports iterators (like in Python) you may do something like:

```
def F():
    a,b = 0,1
    while True:
        yield a
        a, b = b, a + b
```

Once you know how to generate Fibonacci Numbers you just have to cycle trough the numbers and check if they verify the given conditions.

Suppose now you wrote a f(n) that returns the n-th term of the Fibonacci Sequence:

```
def SubFib(startNumber, endNumber):
    n = 0
    cur = f(n)
    while cur <= endNumber:
        if startNumber <= cur:
            print cur
        n += 1
        cur = f(n)
```

or using iterators:

```
def SubFib(startNumber, endNumber):
    for cur in F():
        if cur > endNumber: return
        if cur >= startNumber:
            yield cur

for i in SubFib(10, 200):
    print i
```

Implementation:

PY

```
def F():
    a,b = 0,1
    while True:
        yield a
        a, b = b, a + b

def SubFib(startNumber, endNumber):
    for cur in F():
        if cur > endNumber: return
        if cur >= startNumber:
            yield cur

for i in SubFib(10, 200):
    print i
```

Q9: LIS: Find length of the *longest increasing subsequence (LIS)* in the array. Solve using DP. ☆☆☆

Topics: Dynamic Programming Data Structures Python JavaScript

Problem:

The **longest increasing subsequence problem** is to find a subsequence of a given sequence in which the subsequence's elements are in sorted order, lowest to highest, and in which the subsequence is as long as possible.

Consider:

In the first 16 terms of the binary Van der Corput sequence

```
0, 8, 4, 12, 2, 10, 6, 14, 1, 9, 5, 13, 3, 11, 7, 15
```

a longest increasing subsequence is

```
0, 2, 6, 9, 11, 15.
```

This subsequence has length six; the input sequence has no seven-member increasing subsequences. The longest increasing subsequence in this example is not unique: for instance,

```
0, 4, 6, 9, 11, 15 or
0, 2, 6, 9, 13, 15 or
0, 4, 6, 9, 13, 15
```

are other increasing subsequences of equal length in the same input sequence.

Solution:

- Let's assume the indices of the array are from 0 to N - 1.
- Let's define `DP[i]` to be the length of the LIS (Longest increasing subsequence) which is ending at element with index `i`.

```
DP[0] = 1; // length of LIS for the first element is always 1
int maxLength = 1;
```

- To compute `DP[i]` for each `i > 0` we look at all indices `j < i` and check both:
  - if `DP[j] + 1 > DP[i]` and
  - `array[j] < array[i]` (we want it to be increasing).
  - If this is `true` we can update the current *optimum* for `DP[i]`.

```
for (int i = 1; i < N; i++)
  {
     DP[i] = 1;
     prev[i] = -1;

     for (int j = i - 1; j >= 0; j--)
        if (DP[j] + 1 > DP[i] && array[j] < array[i])
        {
           DP[i] = DP[j] + 1;
           prev[i] = j;
        }

     if (DP[i] > maxLength)
     {
        bestEnd = i;
        maxLength = DP[i];
     }
  }
```

- To find the global *optimum* for the array you can take the maximum value from DP `[0...N - 1]`
- Use the array `prev` to be able later to find the actual sequence not only its length. Just go back recursively from `bestEnd` in a loop using `prev[bestEnd]`. The `-1` value is a sign to stop.

Complexity Analysis:

Time Complexity: O(n^2) **Space Complexity**: O(n^2)

- Time complexity : `O(n^2)`. Two loops of `n` are there.
- Space complexity : `O(n)`. DP array of size `n` is used.

Implementation:

JS

```
/**
 * Dynamic programming approach to find longest increasing subsequence.
 * Complexity: O(n * n)
 *
 * @param {number[]} sequence
 * @return {number}
 */
export default function dpLongestIncreasingSubsequence(sequence) {
  // Create array with longest increasing substrings length and
  // fill it with 1-s that would mean that each element of the sequence
  // is itself a minimum increasing subsequence.
  const lengthsArray = Array(sequence.length).fill(1);

  let previousElementIndex = 0;
  let currentElementIndex = 1;

  while (currentElementIndex < sequence.length) {
    if (sequence[previousElementIndex] < sequence[currentElementIndex]) {
      // If current element is bigger then the previous one then
      // current element is a part of increasing subsequence which
      // length is by one bigger then the length of increasing subsequence
      // for previous element.
      const newLength = lengthsArray[previousElementIndex] + 1;
      if (newLength > lengthsArray[currentElementIndex]) {
        // Increase only if previous element would give us bigger subsequence length
        // then we already have for current element.
        lengthsArray[currentElementIndex] = newLength;
      }
    }

    // Move previous element index right.
    previousElementIndex += 1;

    // If previous element index equals to current element index then
    // shift current element right and reset previous element index to zero.
    if (previousElementIndex === currentElementIndex) {
      currentElementIndex += 1;
      previousElementIndex = 0;
    }
  }

  // Find the biggest element in lengthsArray.
  // This number is the biggest length of increasing subsequence.
  let longestIncreasingLength = 0;

  for (let i = 0; i < lengthsArray.length; i += 1) {
    if (lengthsArray[i] > longestIncreasingLength) {
      longestIncreasingLength = lengthsArray[i];
    }
  }

  return longestIncreasingLength;
}
```

Java

```java
public int longestIncreasingSubsequence(int[] nums) {
    if (nums == null || nums.length == 0) {
        return 0;
    }
    int n = nums.length;
    int[] dp = new int[n];
    int max = 0;
    for (int i = 0; i < n; i++) {
        dp[i] = 1;
        for (int j = 0; j < i; j++) {
            if (nums[j] <= nums[i]) {
                dp[i] = dp[i] > dp[j] + 1 ? dp[i] : dp[j] + 1;
            }
        }
        if (dp[i] > max) {
            max = dp[i];
        }
    }
    return max;
}
```

PY

Traditional DP solution.

```python
# Time:  O(n^2)
# Space: O(n)
# Traditional DP solution.
class SolutionDP(object):
    def lengthOfLIS(self, nums):
        """
        :type nums: List[int]
        :rtype: int
        """
        dp = []  # dp[i]: the length of LIS ends with nums[i]
        for i in xrange(len(nums)):
            dp.append(1)
            for j in xrange(i):
                if nums[j] < nums[i]:
                    dp[i] = max(dp[i], dp[j] + 1)
        return max(dp) if dp else 0
```

Or in 5 lines of code:

```python
def lis(a):
    L = []
    for (k,v) in enumerate(a):
        L.append(max([L[i] for (i,n) in enumerate(a[:k]) if n<v] or [[]], key=len) + [v])
    return max(L, key=len)

inp = [int(a) for a in input("List of integers: ").split(' ')]
print(lis(inp));
```

Q10: How to merge two sorted *Arrays* into a *Sorted Array*? ☆☆☆

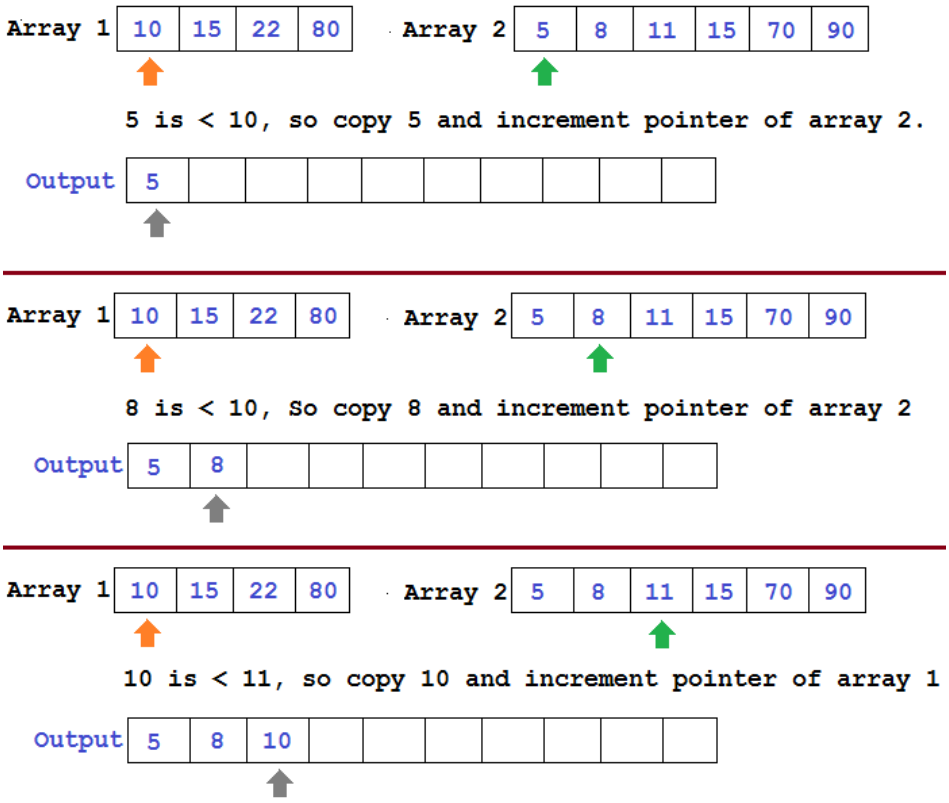
Topics: Arrays Data Structures Java JavaScript Python

Answer:

Let's look at the **merge principle**:

Given two separate lists A and B ordered from least to greatest, construct a list C by:

- repeatedly comparing the least value of A to the least value of B,
- removing (or moving a pointer) the lesser value, and appending it onto C.
- when one list is exhausted, append the remaining items in the other list onto C in order.
- The list C is then also a sorted list.



Complexity Analysis:

Time Complexity: O(n) **Space Complexity**: O(n)

This problem will have `O(n)` complexity at best.

Implementation:

JS

```
function merge(a, b) {
    var result = [];
    var ai = 0;
    var bi = 0;
    while (true) {
        if ( ai < a.length && bi < b.length) {
            if (a[ai] < b[bi]) {
                result.push(a[ai]);
                ai++;
            } else if (a[ai] > b[bi]) {
                result.push(b[bi]);
                bi++;
            } else {
                result.push(a[ai]);
                result.push(b[bi]);
                ai++;
                bi++;
            }
```

```
        } else if (ai < a.length) {
            result.push.apply(result, a.slice(ai, a.length));
            break;
        } else if (bi < b.length) {
            result.push.apply(result, b.slice(bi, b.length));
            break;
        } else {
            break;
        }
    }
    return result;
}
```

Java

```
public static int[] merge(int[] a, int[] b) {

    int[] answer = new int[a.length + b.length];
    int i = 0, j = 0, k = 0;

    while (i < a.length && j < b.length)
       answer[k++] = a[i] < b[j] ? a[i++] :  b[j++];

    while (i < a.length)
        answer[k++] = a[i++];

    while (j < b.length)
        answer[k++] = b[j++];

    return answer;
}
```

PY

```
def merge(left, right):
    result = []
    i ,j = 0, 0
    while i < len(left) and j < len(right):
        print('left[i]: {} right[j]: {}'.format(left[i],right[j]))
        if left[i] <= right[j]:
            print('Appending {} to the result'.format(left[i]))
            result.append(left[i])
            print('result now is {}'.format(result))
            i += 1
            print('i now is {}'.format(i))
        else:
            print('Appending {} to the result'.format(right[j]))
            result.append(right[j])
            print('result now is {}'.format(result))
            j += 1
            print('j now is {}'.format(j))
    print('One of the list is exhausted. Adding the rest of one of the lists.')
    result += left[i:]
    result += right[j:]
    print('result now is {}'.format(result))
    return result
```

Q11: Find all the *Permutations* of a String ☆☆☆

Topics: Backtracking Strings Data Structures Java Python JavaScript

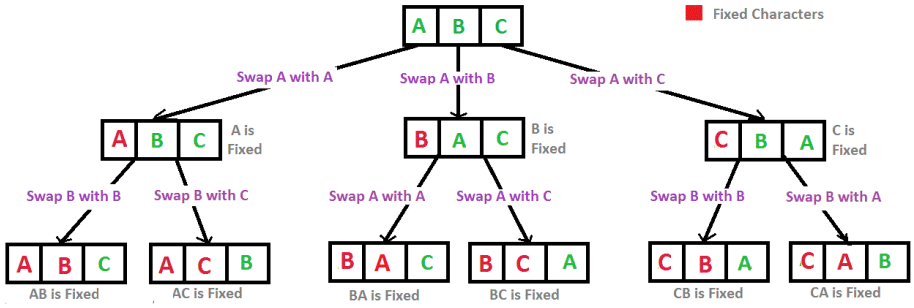
Answer:

A **permutation**, also called an “arrangement number” or “order,” is a rearrangement of the elements of an ordered list S into a one-to-one correspondence with S itself (or in simple english: permutation is each of several possible ways in which a set or number of things can be ordered or arranged). A string of length `n` has `n!` permutation:

```
ABC // original string (3)
ABC ACB BAC BCA CBA CAB // permutations 3! = 6
```

To print all permutations of a string use *backtracking* implemented via *recursion*:

- Try each of the letters in turn as the *first letter* and then find all the permutations of the *remaining letters* using a recursive call.
- The *base case* is when the input is an empty string the only permutation is the empty string.
- In other words, you simply traverse the tree, and when you reach the leaf you print the permutation. Then you backtrack one level up, and try another option. Moving one level up the tree is what we call the backtracking in this case.



Recursion Tree for Permutations of String "ABC"

Complexity Analysis:

Time Complexity: O(n!) **Space Complexity**: O(n!)

For any given string of length `n` there are `n!` possible permutations, and we need to print all of them so Time complexity is `O(n * n!)`. The function will be called recursively and will be stored in call stack for all `n!` permutations, so Space complexity is `O(n!)`.

Implementation:

JS

```
// using backtracking
let permute = (str, left = 0, right = str.length - 1) => {
  //If left index is equal to right index
  //Print the string permutation
  if(left == right){
    console.log(str);
  }else{
    for(let i = left; i <= right; i++){
      //Swap the letters of the string
      str = swap(str, left, i);
      //Generate the permutation with swapped letters
      permute(str, left+1, right);
      //Restore the letters back to their position, e.g. backtrack to prev state of the string
      str = swap(str, left, i);
    }
  }
}
```

```
//Function to swap the letters of the string
let swap = (str, left, right) => {
  let arr = str.split('');
  [arr[left], arr[right]] = [arr[right], arr[left]];
  return arr.join('');
}
```

Java

```
public static void permutation(String str) {
    permutation("", str);
}

private static void permutation(String prefix, String str) {
    int n = str.length();
    if (n == 0) System.out.println(prefix);
    else {
        for (int i = 0; i < n; i++)
            // take new prefix from string[i], put it as a first element and construct the rest of the
string
            permutation(prefix + str.charAt(i), str.substring(0, i) + str.substring(i+1, n));
    }
}
```

PY

```
# using backtracking
def permute(a, l, r):
    if l==r:
        print toString(a)
    else:
        for i in xrange(l,r+1):
            a[l], a[i] = a[i], a[l]
            permute(a, l+1, r)
            a[l], a[i] = a[i], a[l] # backtrack

# Driver program to test the above function
string = "ABC"
n = len(string)
a = list(string)
permute(a, 0, n-1)
```

Q12: How to check if String is a Palindrome? ☆☆☆

Topics: Strings Python

Answer:

A **palindrome** is a word, phrase, number or other sequence of units that *can be read the same way in either direction*. You can check if a string is a palindrome by comparing it to the reverse of itself either using simple *loop* or *recursion*.

Complexity Analysis:

Time Complexity: O(n) **Space Complexity**: O(n)

The time complexity is `O(n/2)` that is still $O(n)$. We doing `n` comparisons, where n = `s.length()`. Each comparison takes $O(1)$ time, as it is a single character comparison. You can cut the complexity of the function in

half by stopping at `i == (s.length() / 2)+1`. It is not relevant in Big O terms, but it is still a pretty decent gain. Loop approach has constant space complexity `O(1)` as we not allocating any new memory. Reverse string approach needs `O(n)` additional memory to create a reversed copy of a string. Recursion approach has `O(n)` space complexity due call stack.

Implementation:

Java

Loop:

```
boolean isPalindrome(String str) {
    int n = str.length();
    for( int i = 0; i < n/2; i++ )
        if (str.charAt(i) != str.charAt(n-i-1)) return false;
    return true;
}
```

Recursion:

```
private boolean isPalindrome(String s) {
    int length = s.length();
    if (length < 2) return true;
    return s.charAt(0) != s.charAt(length - 1) ? false :
            isPalindrome(s.substring(1, length - 1));
}
```

You may want to solve the problem by *inverting* the original string in a whole but note the space complexity will be worst than for the loop solution (`O(n)` instead of `O(1)`):

```
public static boolean isPalindrome(String str) {
    return str.equals(new StringBuilder(str).reverse().toString());
}
```

PY

```
str(n) == str(n)[::-1]
```

Explanation:

- We're checking if the string representation of `n` equals the inverted string representation of `n`
- The `[::-1]` slice takes care of inverting the string
- After that, we compare for equality using `==`

Q13: Write a program for Recursive Binary Search ☆☆☆

Topics: Searching Java Python JavaScript

Answer:

- The first difference is that the while loop from iterative approach is replaced by a recursive call back to the same method with the new values of `low` and `high` passed to the next recursive invocation along with

`Array` and `key` or target element.

- The second difference is that instead of returning false when the while loop exits in the iterative version, in case of the recursive version, the condition of `low > high` is checked at the beginning of the next level of recursion and acts as the boundary condition for stopping further recursive calls by returning false.
- Also, note that the recursive invocations of `binarySearch()` return back the search result up the recursive call stack so that true or false return value is passed back up the call stack without any further processing.
- To further optimize the solution, in term of running time, we could consider implement the **tail-recursive solution**, where the stack trace of the algorithm would not pile up, which leads to a less memory footprint during the running time. To have the tail-recursive solution, the trick is to simply return the result from the next recursive function instead of further processing.

Implementation:

JS

```
function binarySearch(sortedArray, searchValue, minIndex, maxIndex) {
    var currentIndex;
    var currentElement;

    while (minIndex <= maxIndex) {
        // Find the value of the middle of the array
        var middleIndex = (minIndex + maxIndex) / 2 | 0;
        currentElement = sortedArray[middleIndex];

        // It's the same as the value in the middle - we can return!
        if (currentElement === searchValue)
        {
          return middleIndex;
        }
        // Is the value less than the value in the middle of the array
        if (currentElement < searchValue) {
          return binarySearch(sortedArray, searchValue, middleIndex + 1, maxIndex);
        }
        // Is the value greater than the value in the middle of the array
        if (currentElement > searchValue) {
          return binarySearch(sortedArray, searchValue, minIndex, middleIndex - 1);
        }
    }

    return -1;
}
```

Java

```
// Find out if a key x exists in the sorted array
// A[low..high] or not using binary search algorithm
public static int binarySearch(int[] A, int low, int high, int x) {
    // Base condition (search space is exhausted)
    if (low > high) {
        return -1;
    }

    // we find the mid value in the search space and
    // compares it with key value

    int mid = (low + high) / 2;

    // overflow can happen. Use beleft
    // int mid = low + (high - low) / 2;

    // Base condition (key value is found)
    if (x == A[mid]) {
        return mid;
    }
```

```
    // discard all elements in the right search space
    // including the mid element
    else if (x < A[mid]) {
        return binarySearch(A, low, mid - 1, x);
    }

    // discard all elements in the left search space
    // including the mid element
    else {
        return binarySearch(A, mid + 1, high, x);
    }
}
```

PY

```
class Solution:
    def searchInsert(self, nums, target):
        return self.tail_recursive_searchInsert(0, nums, target)


    def tail_recursive_searchInsert(self, start, nums, target):
        """
        :type nums: List[int]
        :type target: int
        :rtype: int

          return the position to insert a new element into the sorted list.
          Note: the solution can be implemented in a recursive way.
        """
        size = len(nums)
        if (size == 0):
            # empty input list, early exit
            return start

        mid_index = int(size/2)
        if (nums[mid_index] == target):
            return start + mid_index
        elif (nums[mid_index] > target):
            return self.tail_recursive_searchInsert(
                        start, nums[0:mid_index], target)
        else:
            return self.tail_recursive_searchInsert(
                        start+mid_index+1, nums[mid_index+1:], target)
```

Q14: Merge two sorted singly Linked Lists without creating new nodes ☆☆☆

Topics: Linked Lists Python

Problem:

You have two singly linked lists that are already sorted, you have to merge them and return a the head of the new list without creating any new extra nodes. The returned list should be sorted as well.

The method signature is:

```
Node MergeLists(Node list1, Node list2);
```

Node class is below:

```
class Node{
   int data;
   Node next;
}
```

Example:

```
Input: 1->2->4, 1->3->4
Output: 1->1->2->3->4->4
```

Solution:

Here is the algorithm on how to merge two sorted linked lists A and B:

```
while A not empty or B not empty:
   if first element of A < first element of B:
      remove first element from A
      insert element into C
   end if
   else:
      remove first element from B
      insert element into C
end while
```

Here C will be the output list.

There are two solutions possible:

- **Recursive approach** is
  i. Of the two nodes passed, keep the node with smaller value
  ii. Point the smaller node's next to the next smaller value of remaining of the two linked lists
  iii. Return the current node (which was smaller of the two nodes passed to the method)

- **Iterative approach** is better for all practical purposes as scales as size of lists swells up:
  i. Treat one of the sorted linked lists as list, and treat the other sorted list as `bag of nodes'.
  ii. Then we lookup for correct place for given node (from bag of nodes) in the list.

Complexity Analysis:

Time Complexity: O(n)

The run time complexity for the recursive and iterative solution here or the variant is $O(n)$.

Implementation:

Java

Recursive solution:

```
Node MergeLists(Node list1, Node list2) {
  if (list1 == null) return list2;
  if (list2 == null) return list1;

  if (list1.data < list2.data) {
    list1.next = MergeLists(list1.next, list2);
```

```
    return list1;
  } else {
    list2.next = MergeLists(list2.next, list1);
    return list2;
  }
}
```

Recursion should not be needed to avoid allocating a new node:

```
Node MergeLists(Node list1, Node list2) {
  if (list1 == null) return list2;
  if (list2 == null) return list1;

  Node head;
  if (list1.data < list2.data) {
    head = list1;
  } else {
    head = list2;
    list2 = list1;
    list1 = head;
  }
  while(list1.next != null) {
    if (list1.next.data > list2.data) {
      Node tmp = list1.next;
      list1.next = list2;
      list2 = tmp;
    }
    list1 = list1.next;
  }
  list1.next = list2;
  return head;
}
```

PY

Recursive:

```
def merge_lists(h1, h2):
    if h1 is None:
        return h2
    if h2 is None:
        return h1

    if (h1.value < h2.value):
        h1.next = merge_lists(h1.next, h2)
        return h1
    else:
        h2.next = merge_lists(h2.next, h1)
        return h2
```

Iterative:

```
def merge_lists(head1, head2):
    if head1 is None:
        return head2
    if head2 is None:
        return head1

    # create dummy node to avoid additional checks in loop
    s = t = node()
    while not (head1 is None or head2 is None):
        if head1.value < head2.value:
            # remember current low-node
            c = head1
```

```
            # follow ->next
            head1 = head1.next
        else:
            # remember current low-node
            c = head2
            # follow ->next
            head2 = head2.next

        # only mutate the node AFTER we have followed ->next
        t.next = c
        # and make sure we also advance the temp
        t = t.next

    t.next = head1 or head2

    # return tail of dummy node
    return s.next
```

Q15: Split the Linked List into `k` consecutive linked list "parts" ☆☆☆

Topics: Linked Lists Python

Problem:

Given a (singly) linked list with head node root, write a function to split the linked list into k consecutive linked list "parts".

- The length of each part should be as equal as possible: no two parts should have a size differing by more than 1. This may lead to some parts being null.
- The parts should be in order of occurrence in the input list, and parts occurring earlier should always have a size greater than or equal parts occurring later.
- Return a List of ListNode's representing the linked list parts that are formed

Example:

```
[1,2,3,4,5,6,7,8,9,10], k=3
ans = [ [1,2,3,4] [5,6,7] [8,9,10] ]
```

Solution:

- If there are `N` nodes in the linked list `root`, then there are `N/k` items in each part, plus the first `N%k` parts have an extra item. We can count `N` with a simple loop.
- Now for each part, we have calculated how many nodes that part will have: `width + (i < remainder ? 1 : 0)`. We create a new list and write the part to that list.

Implementation:

Java

```
public ListNode[] splitListToParts(ListNode root, int k) {
  ListNode cur = root;
  int N = 0;
  while (cur != null) {
    cur = cur.next;
    N++;
  }
```

```
  int width = N / k, rem = N % k;

  ListNode[] ans = new ListNode[k];
  cur = root;
  for (int i = 0; i < k; ++i) {
    ListNode head = new ListNode(0), write = head;
    for (int j = 0; j < width + (i < rem ? 1 : 0); ++j) {
      write = write.next = new ListNode(cur.val);
      if (cur != null) cur = cur.next;
    }
    ans[i] = head.next;
  }
  return ans;
}
```

PY

```
def splitListToParts(self, root, k):
    cur = root
    for N in xrange(1001):
        if not cur: break
        cur = cur.next
    width, remainder = divmod(N, k)

    ans = []
    cur = root
    for i in xrange(k):
        head = write = ListNode(None)
        for j in xrange(width + (i < remainder)):
            write.next = write = ListNode(cur.val)
            if cur: cur = cur.next
        ans.append(head.next)
    return ans
```

Q16: Check if parentheses are balanced using Stack ☆☆☆

Topics: Stacks Java Python JavaScript

Answer:

One of the most important applications of stacks is to check if the parentheses are balanced in a given expression. The compiler generates an error if the parentheses are not matched.

Here are some of the *balanced* and *unbalanced* expressions:

BALANCED EXPRESSION	UNBALANCED EXPRESSION
(a + b)	(a + b
[(c – d) * e]	[(c – d * e]
{ ( ) } [ ]	{ [( ]) }

Algorithm

1. Declare a character stack which will hold an array of all the opening parenthesis.
2. Now traverse the expression string exp.

3. If the current character is a starting bracket (`(` or `{` or `[`) then push it to stack.
4. If the current character is a closing bracket (`)` or `}` or `]`) then pop from stack and if the popped character is the matching starting bracket then fine else parenthesis are not balanced.
5. After complete traversal, if there is some starting bracket left in stack then “not balanced”

Complexity Analysis:

Time Complexity: O(n) **Space Complexity**: O(n)

- We are traversing through each character of string, Time complexity $O(n)$.
- We are storing the opposite parentheses characters in the stack, in the worst case there can be all the opposite characters in the string, Space complexity $O(n)$.

Implementation:

JS

```
let isMatchingBrackets = function (str) {
    let stack = [];
    let map = {
        '(': ')',
        '[': ']',
        '{': '}'
    }

    for (let i = 0; i < str.length; i++) {

        // If character is an opening brace add it to a stack
        if (str[i] === '(' || str[i] === '{' || str[i] === '[' ) {
            stack.push(str[i]);
        }
        //  If that character is a closing brace, pop from the stack, which will also reduce the length
of the stack each time a closing bracket is encountered.
        else {
            let last = stack.pop();

            //If the popped element from the stack, which is the last opening brace doesn’t match the
corresponding closing brace in the map, then return false
            if (str[i] !== map[last]) {return false};
        }
    }
    // By the completion of the for loop after checking all the brackets of the str, at the end, if the
stack is not empty then fail
        if (stack.length !== 0) {return false};

    return true;
}
```

Java

```
public static boolean balancedParenthensies(String s) {
    Stack<Character> stack  = new Stack<Character>();
    for(int i = 0; i < s.length(); i++) {
        char c = s.charAt(i);
        if(c == '[' || c == '(' || c == '{' ) {
            stack.push(c);
        } else if(c == ']') {
            if(stack.isEmpty() || stack.pop() != '[') {
                return false;
            }
        } else if(c == ')') {
            if(stack.isEmpty() || stack.pop() != '(') {
                return false;
```

```
            }
        } else if(c == '}') {
            if(stack.isEmpty() || stack.pop() != '{') {
                return false;
            }
        }

    }
    return stack.isEmpty();
}
```

PY

```
open_list = ["[","{","("]
close_list = ["]","}",")"]

# Function to check parentheses
def check(myStr):
    stack = []
    for i in myStr:
        if i in open_list:
            stack.append(i)
        elif i in close_list:
            pos = close_list.index(i)
            if ((len(stack) > 0) and
                (open_list[pos] == stack[len(stack)-1])):
                stack.pop()
            else:
                return "Unbalanced"
    if len(stack) == 0:
        return "Balanced"
    else:
        return "Unbalanced"
```

Q17: Sum two numbers represented as Linked Lists ☆☆☆

Topics: Linked Lists Python

Problem:

You are given two **non-empty** linked lists representing two non-negative integers. The digits are stored in **reverse order** and each of their nodes contain a single digit. Add the two numbers and return it as a linked list.

You may assume the two numbers do not contain any leading zero, except the number 0 itself.

Example:

```
Input: (2 -> 4 -> 3) + (5 -> 6 -> 4)
Output: 7 -> 0 -> 8
Explanation: 342 + 465 = 807.
```

Solution:

You can add two numbers represented using LinkedLists in the same way you add two numbers by hand.

Iterate over the linked lists, add their corresponding elements, keep a carry just the way you do while adding numbers by hand, add the carry-over from the previous addition to the current sum and so on.

One of the trickiest parts of this problem is the issue of the carried number - if every pair of nodes added to a number less than 10, then there wouldn't be a concern of 'carrying' digits over to the next node. However, adding numbers like 4 and 6 produces a carry.

If one list is longer than the other, we still will want to add the longer list's nodes to the solution, so we'll have to make sure we continue to check so long as the nodes aren't null. That means the while loop should keep going as long as list 1 isn't `null` OR list 2 isn't `null`.

Implementation:

Java

```
public ListNode addTwoNumbers(ListNode l1, ListNode l2) {
    ListNode head = new ListNode(0);
    ListNode result = head;
    int carry = 0;

    while (l1 != null || l2 != null || carry > 0) {
        int resVal = (l1 != null? l1.val : 0) + (l2 != null? l2.val : 0) + carry;
        result.next = new ListNode(resVal % 10);
        carry = resVal / 10;
        l1 = (l1 == null ? l1 : l1.next);
        l2 = (l2 == null ? l2 : l2.next);
        result = result.next;
    }

    return head.next;
}
```

PY

```
class Solution:
# @return a ListNode
def addTwoNumbers(self, l1, l2):
    carry = 0
    root = n = ListNode(0)
    while l1 or l2 or carry:
        v1 = v2 = 0
        if l1:
            v1 = l1.val
            l1 = l1.next
        if l2:
            v2 = l2.val
            l2 = l2.next
        carry, val = divmod(v1+v2+carry, 10)
        n.next = ListNode(val)
        n = n.next
    return root.next
```

Q18: How to make *a chain of function decorators*? ☆☆☆☆

Topics: Python

Problem:

How can I make two decorators in Python that would do the following?

```
@makebold
@makeitalic
```

```
def say():
   return "Hello"
```

which should return:

```
"<b><i>Hello</i></b>"
```

Solution:

Consider:

```
from functools import wraps

def makebold(fn):
    @wraps(fn)
    def wrapped(*args, **kwargs):
        return "<b>" + fn(*args, **kwargs) + "</b>"
    return wrapped

def makeitalic(fn):
    @wraps(fn)
    def wrapped(*args, **kwargs):
        return "<i>" + fn(*args, **kwargs) + "</i>"
    return wrapped

@makebold
@makeitalic
def hello():
    return "hello world"

@makebold
@makeitalic
def log(s):
    return s

print hello()        # returns "<b><i>hello world</i></b>"
print hello.__name__ # with functools.wraps() this returns "hello"
print log('hello')   # returns "<b><i>hello</i></b>"
```

Q19: Explain how Radix Sort works ☆☆☆☆

Topics: Sorting Python JavaScript

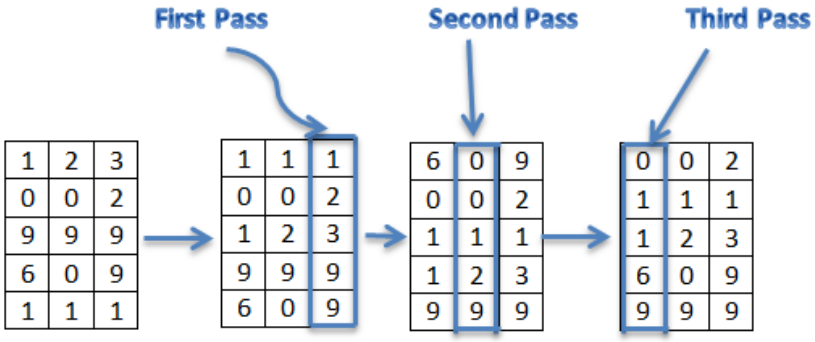
Answer:

Radix sort is a sorting technique that sorts the elements by first grouping the individual digits of the same place value. Then, sort the elements according to their increasing/decreasing order. Radix sort uses **counting sort** as a subroutine to sort an array of numbers. Because integers can be used to represent strings (by hashing the strings to integers), radix sort works on data types other than just integers.

Radix sort is a *non-comparative* sorting algorithm. It avoids comparison by creating and distributing elements into buckets according to their *radix*. For example, a binary system (using numbers 0 and 1) has a radix of 2 and a decimal system (using numbers 0 to 9) has a radix of 10.

Radix sorts can be implemented to start at either the most significant digit (**MSD**) or least significant digit (**LSD**). For example, with `1234`, one could start with `1` (MSD) or `4` (LSD).

Assume the input array is `[123, 2, 999, 609, 111]`:



1. Based on the algorithm, we will sort the input array according to the one's digit (least significant digit, LSD):

```
[0]:
[1]: 111
[2]: 002
[3]: 123
[4]:
[5]:
[6]:
[7]:
[8]:
[9]: 999, 609 //  Note that we placed 999 before 609 because it appeared first.
```

2. Now, we'll sort according to the ten's digit:

```
[0]: 609, 002
[1]: 111
[2]: 123
[3]:
[4]:
[5]:
[6]:
[7]:
[8]:
[9]: 999
```

3. Finally , we sort according to the hundred's digit (most significant digit):

```
[0]: 002
[1]: 111, 123
[2]:
[3]:
[4]:
[5]:
[6]: 609
[7]:
[8]:
[9]: 999
```

Complexity Analysis:

Time Complexity: O(n) **Space Complexity**: O(n)

Radix sort will operate on `n` d-digit numbers where each digit can be one of at most `b` different values (since `b` is the base (or radix) being used). For example, in base 10, a digit can be `0`, `1`, `2`, `3`, `4`, `5`, `6`, `7`, `8` or `9`.

Radix sort uses counting sort on each digit. Each pass over `n` `d`-digit numbers will take `O(n + b)` time, and there are `d` passes total. Therefore, the total running time of radix sort is `O(d(n+b))`. When `d` is a constant and `b` isn't much larger than `n` (in other words, `b=O(n)`), then radix sort takes **linear time**.

The space complexity comes from counting sort, which requires `O(n+b)` space to hold the counts, indices, and output lists.

Implementation:

JS

```
/* Radix Sort implementation in JavaScript */

function radixSort(arr) {
    var maxDigit = getMaxDigitLen(arr);

    //Looping for every digit index from leftmost digits to rightmost digits
    for (var i = 1; i <= maxDigit; i++) {

        // rebuild the digit buckets according to this digit
        var digitBuckets = [];
        for (var j = 0; j < arr.length; j++) {
            var digit = getDigit(arr[j], i); //get the i-th digit of the j-th element of the array

            digitBuckets[digit] = digitBuckets[digit] || []; //If empty initialize with empty array
            digitBuckets[digit].push(arr[j]); //Add the number in it's respective digits bucket
        }

        // rebuild the array according to this digit
        var index = 0;
        for (var k = 0; k < digitBuckets.length; k++) {
            if (digitBuckets[k] && digitBuckets[k].length > 0) {
                for (j = 0; j < digitBuckets[k].length; j++) {
                    arr[index++] = digitBuckets[k][j];
                }
            }
        }
    }
    return arr;
}

// helper function to get the last n-th(position) digit of a number
var getDigit = function(num, position) {
    var num_length = num.toString().length;

    if (num_length < position)
        return 0;

    var radixPosition = Math.pow(10, position - 1); //number with the positions-th power of 10

    /*
    Logic: To find the 2nd digit of the number 123
           we can divide it by 10, which is 12
           and the using the modulus operator(%) we can find 12 % 10 = 2
    */
    var digit = (Math.floor(num / radixPosition)) % 10;

    return digit;
};

// helper function get the length of digits of the max value in this array
var getMaxDigitLen = function(arr) {
    var maxVal = Math.max.apply(Math, arr); //Finding max value from the array

    return Math.ceil(Math.log10(maxVal));
};
```

PY

```
import math

def counting_sort(A, digit, radix):
    #"A" is a list to be sorted, radix is the base of the number system, digit is the digit
    #we want to sort by

    #create a list B which will be the sorted list
    B = [0]*len(A)
    C = [0]*int(radix)
    #counts the number of occurences of each digit in A
    for i in range(0, len(A)):
        digit_of_Ai = (A[i]/radix**digit)%radix
        C[digit_of_Ai] = C[digit_of_Ai] +1
        #now C[i] is the value of the number of elements in A equal to i

    #this FOR loop changes C to show the cumulative # of digits up to that index of C
    for j in range(1,radix):
        C[j] = C[j] + C[j-1]
        #here C is modifed to have the number of elements <= i
    for m in range(len(A)-1, -1, -1): #to count down (go through A backwards)
        digit_of_Ai = (A[m]/radix**digit)%radix
        C[digit_of_Ai] = C[digit_of_Ai] -1
        B[C[digit_of_Ai]] = A[m]

    return B

def radix_sort(A, radix):
    #radix is the base of the number system
    #k is the largest number in the list
    k = max(A)
    #output is the result list we will build
    output = A
    #compute the number of digits needed to represent k
    digits = int(math.floor(math.log(k, radix)+1))
    for i in range(digits):
        output = counting_sort(output,i,radix)

    return output
```

Q20: Given a singly Linked List, determine if it is a Palindrome

☆☆☆☆

Topics: Linked Lists Java Python

Problem:

Could you do it in `O(1)` time and `O(1)` space?

Solution:

Solution: is **Reversed first half **== **Second half**?

Let's look to an example `[1, 1, 2, 1]`. In the beginning, set two pointers `fast` and `slow` starting at the head.

```
1 -> 1 -> 2 -> 1 -> null
sf
```

(1) **Move:** `fast` pointer goes to the end, and `slow` goes to the middle.

```
1 -> 1 -> 2 -> 1 -> null
          s          f
```

(2) **Reverse:** the right half is reversed, and `slow` pointer becomes the 2nd head.

```
1 -> 1    null <- 2 <- 1
h                     s
```

(3) **Compare:** run the two pointers `head` and `slow` together and compare.

```
1 -> 1    null <- 2 <- 1
     h            s
```

Implementation:

Java

```
public boolean isPalindrome(ListNode head) {
    ListNode fast = head, slow = head;
    while (fast != null && fast.next != null) {
        fast = fast.next.next;
        slow = slow.next;
    }
    if (fast != null) { // odd nodes: let right half smaller
        slow = slow.next;
    }
    slow = reverse(slow);
    fast = head;

    while (slow != null) {
        if (fast.val != slow.val) {
            return false;
        }
        fast = fast.next;
        slow = slow.next;
    }
    return true;
}

public ListNode reverse(ListNode head) {
    ListNode prev = null;
    while (head != null) {
        ListNode next = head.next;
        head.next = prev;
        prev = head;
        head = next;
    }
    return prev;
}
```

PY

```
# Note: while comparing the two halves, restore the list to its original
# state by reversing the first half back.

def isPalindrome(self, head):
```

```
    rev = None
    fast = head
    while fast and fast.next:
        fast = fast.next.next
        rev, rev.next, head = head, rev, head.next
    tail = head.next if fast else head
    isPali = True
    while rev:
        isPali = isPali and rev.val == tail.val
        head, head.next, rev = rev, head, rev.next
        tail = tail.next
    return isPali
```

FullStack.Cafe - Kill Your Tech Interview

Q1: Create function that similar to `os.walk` ☆☆☆☆

Topics: Python

Answer:

```
# function that similar to `os.walk`
def print_directory_contents(sPath):
    import os
    for sChild in os.listdir(sPath):
        sChildPath = os.path.join(sPath,sChild)
        if os.path.isdir(sChildPath):
            print_directory_contents(sChildPath)
        else:
            print(sChildPath)
```

Q2: What will be *returned* by this code? ☆☆☆☆

Topics: Python

Problem:

Consider:

```
>>> squares = []
>>> for x in range(5):
...     squares.append(lambda: x**2)
>>> squares[2]()
>>> squares[4]()
```

Solution:

It will return:

```
>>> squares[2]()
16
>>> squares[4]()
16
```

This happens because `x` is not local to the lambdas, but is defined in the outer scope, and it is accessed when the lambda is called — not when it is defined. At the end of the loop, the value of `x` is 4, so all the functions now return 4**2, i.e. 16.

Q3: How do I write a function with *output parameters (call by reference)* ☆☆☆☆

Topics: Python

Answer:

In Python arguments are passed *by assignment*. When you call a function with a parameter, a **new reference** is created that refers to the object passed in. This is separate from the reference that was used in the function call, so there's no way to update that reference and make it refer to a new object.

If you pass a *mutable* object into a method, the method gets a reference to that same object and you can mutate it to your heart's delight, but if you rebind the reference in the method (like `b = b + 1`), the outer scope will know nothing about it, and after you're done, the outer reference will still point at the original object.

So to achieve the desired effect your best choice is to return a tuple containing the multiple results:

```
def func2(a, b):
    a = 'new-value'        # a and b are local names
    b = b + 1              # assigned to new objects
    return a, b            # return new values

x, y = 'old-value', 99
x, y = func2(x, y)
print(x, y)                # output: new-value 100
```

Q4: Whenever you *exit* Python, is all *memory de-allocated*? ☆☆☆☆

Topics: Python

Answer:

The answer here is **no**.

- The modules with *circular references to other objects*, or to objects referenced from global namespaces, aren't always freed on exiting Python.
- Plus, it is impossible to de-allocate *portions of memory reserved by the C library*.

Q5: Show me *three different ways* of fetching *every third item* in the list ☆☆☆☆

Topics: Python

Answer:

```
# 1
[x for i, x in enumerate(thelist) if i%3 == 0]

# 2
for i, x in enumerate(thelist):
    if i % 3: continue
    yield x

# 3
a = 0
for x in thelist:
    if a%3: continue
    yield x
    a += 1
```

Q6: What will be the *output of the code* below? ☆☆☆☆

Topics: Python

Problem:

Consider:

```
list = ['a', 'b', 'c', 'd', 'e']
print list[10:]
```

Solution:

The above code will output `[]`, and will not result in an `IndexError`.

As one would expect, attempting to access a member of a list using an index that exceeds the number of members (e.g., attempting to access `list[10]` in the list above) results in an `IndexError`. However, attempting to access a slice of a list at a starting index that exceeds the number of members in the list will not result in an `IndexError` and will simply return an empty list.

What makes this a particularly nasty gotcha is that it can lead to bugs that are really hard to track down since no error is raised at runtime.

Q7: Will the code below work? *Why* or why not? ☆☆☆☆

Topics: Python

Problem:

Given the following subclass of dictionary:

```
class DefaultDict(dict):
  def __missing__(self, key):
    return []
```

Will the code below work? Why or why not?

```
d = DefaultDict()
d['florp'] = 127
```

Solution:

Actually, the code shown will work with the standard dictionary object in python 2 or 3—that is normal behavior. Subclassing dict is unnecessary. However, the subclass still won't work with the code shown because `__missing__` returns a value but does not change the dict itself:

```
d = DefaultDict()
print d
{}
print d['foo']
[]
print d
{}
```

So it will “work,” in the sense that it won’t produce any error, but doesn’t do what it seems to be intended to do.

Here is a `__missing__`-based method that will update the dictionary, as well as return a value:

```
class DefaultDict(dict):
    def __missing__(self, key):
        newval = []
        self[key] = newval
        return newval
```

But since version 2.5, a defaultdict object has been available in the collections module (in the standard library.)

Q8: What will this code *return*? ☆☆☆☆☆

Topics: Python

Problem:

Consider:

```
a = True
print(('A', 'B')[a == False])
```

What will this code return? Explain.

Solution:

Here `('A', 'B')` is a tuple. We could access values in tuple, use the square brackets `[]`. The `a == False` is an expression that could be evaluated as boolean. In Python 3.x `True` and `False` are keywords and will always be equal to 1 and 0. So the result will be `A`:

```
a = True
print(('A', 'B')[a == False])
# (falseValue, trueValue)[test]
# Output: A
```

Q9: How to read a `8GB` file in Python? ☆☆☆☆☆

Topics: Python

Answer:

All you need to do is use the file object as an *iterator*.

```
for line in open("log.txt"):
    do_something_with(line)
```

Even better is using `context manager` in recent Python versions.

```
with open("log.txt") as fileobject:
    for line in fileobject:
        do_something_with(line)
```

This will automatically close the file as well.

Q10: How should one access *nonlocal variables in closures* in python 2.x? ☆☆☆☆☆

Topics: Python

Answer:

Inner functions can *read* nonlocal variables in 2.x, just not *rebind* them. This is annoying, but you can work around it. Just create a dictionary, and store your data as elements therein. Inner functions are not prohibited from *mutating* the objects that nonlocal variables refer to.

```
def outer():
    d = {'y' : 0}
    def inner():
        d['y'] += 1
        return d['y']
    return inner

f = outer()
print(f(), f(), f()) #prints 1 2 3
```

Or using **nonlocal class**:

```
def outer():
    class context:
        y = 0
    def inner():
        context.y += 1
        return context.y
    return inner
```

It works because you can *modify* nonlocal variables. But you cannot do *assignment* to nonlocal variables.

Q11: How do I *access a module* written in Python *from C*? ☆☆☆☆☆

Topics: Python

Answer:

You can get a pointer to the module object by calling `PyImport_ImportModule`:

```
module = PyImport_ImportModule("mymodule");
```

You can then access the module's attributes (i.e. any name defined in the module) as follows:

```
attr = PyObject_GetAttrString(module, "name");
```

Calling `PyObject_SetAttrString` to assign to variables in the module also works.