To understand what `yield` does, you must understand what *generators* are. And before generators come *iterables*.

Iterables

When you create a list, you can read its items one by one. Reading its items one by one is called iteration:

```
>>> mylist = [1, 2, 3]
>>> for i in mylist:
...    print(i)
1
2
3
```

`mylist` is an *iterable*. When you use a list comprehension, you create a list, and so an iterable:

```
>>> mylist = [x*x for x in range(3)]
>>> for i in mylist:
...    print(i)
0
1
4
```

Everything you can use " `for... in...` " on is an iterable; `lists` , `strings` , files...

These iterables are handy because you can read them as much as you wish, but you store all the values in memory and this is not always what you want when you have a lot of values.

Generators

Generators are iterators, but **you can only iterate over them once**. It's because they do not store all the values in memory, **they generate the values on the fly**:

```
>>> mygenerator = (x*x for x in range(3))
>>> for i in mygenerator:
...    print(i)
0
1
4
```

It is just the same except you used `()` instead of `[]` . BUT, you **cannot** perform `for i in mygenerator` a second time since generators can only be used once: they calculate 0, then forget about it and calculate 1, and end calculating 4, one by one.

Yield

`Yield` is a keyword that is used like `return` , except the function will return a generator.

```
>>> def createGenerator():
...    mylist = range(3)
...    for i in mylist:
...        yield i*i
...
>>> mygenerator = createGenerator() # create a generator
>>> print(mygenerator) # mygenerator is an object!
<generator object createGenerator at 0xb7555c34>
>>> for i in mygenerator:
...     print(i)
0
1
4
```

Here it's a useless example, but it's handy when you know your function will return a huge set of values that you will only need to read once.

To master `yield`, you must understand that **when you call the function, the code you have written in the function body does not run.** The function only returns the generator object, this is a bit tricky :-)

Then, your code will be run each time the `for` uses the generator.

Now the hard part:

The first time the `for` calls the generator object created from your function, it will run the code in your function from the beginning until it hits `yield`, then it'll return the first value of the loop. Then, each other call will run the loop you have written in the function one more time, and return the next value, until there is no value to return.

The generator is considered empty once the function runs but does not hit yield anymore. It can be because the loop had come to an end, or because you do not satisfy a `"if/else"` anymore.

Your code explained

Generator:

```
# Here you create the method of the node object that will return the generator
def node._get_child_candidates(self, distance, min_dist, max_dist):

  # Here is the code that will be called each time you use the generator object:

  # If there is still a child of the node object on its left
  # AND if distance is ok, return the next child
  if self._leftchild and distance - max_dist < self._median:
      yield self._leftchild

  # If there is still a child of the node object on its right
  # AND if distance is ok, return the next child
  if self._rightchild and distance + max_dist >= self._median:
      yield self._rightchild

  # If the function arrives here, the generator will be considered empty
  # there is no more than two values: the left and the right children
```

Caller:

```
# Create an empty list and a list with the current object reference
result, candidates = list(), [self]

# Loop on candidates (they contain only one element at the beginning)
while candidates:

    # Get the last candidate and remove it from the list
    node = candidates.pop()

    # Get the distance between obj and the candidate
    distance = node._get_dist(obj)

    # If distance is ok, then you can fill the result
    if distance <= max_dist and distance >= min_dist:
        result.extend(node._values)

    # Add the children of the candidate in the candidates list
    # so the loop will keep running until it will have looked
    # at all the children of the children of the children, etc. of the candidate
    candidates.extend(node._get_child_candidates(distance, min_dist, max_dist))

return result
```

This code contains several smart parts:

- The loop iterates on a list but the list expands while the loop is being iterated :-) It's a concise way to go through all these nested data even if it's a bit dangerous since you can end up with an infinite loop. In this case, `candidates.extend(node._get_child_candidates(distance, min_dist, max_dist))` exhausts all the values of the generator, but `while` keeps creating new generator objects which will produce different values from the previous ones since it's not applied on the same node.
- The `extend()` method is a list object method that expects an iterable and adds its values to the list.

Usually we pass a list to it:

```
>>> a = [1, 2]
>>> b = [3, 4]
>>> a.extend(b)
>>> print(a)
[1, 2, 3, 4]
```

But in your code it gets a generator, which is good because:

1. You don't need to read the values twice.
2. You can have a lot of children and you don't want them all stored in memory.

And it works because Python does not care if the argument of a method is a list or not. Python expects iterables so it will work with strings, lists, tuples and generators! This is called duck typing and is one of the reason why Python is so cool. But this is another story, for another question...

You can stop here, or read a little bit to see a advanced use of generator:

Controlling a generator exhaustion

```
>>> class Bank(): # let's create a bank, building ATMs
...     crisis = False
...     def create_atm(self):
...         while not self.crisis:
...             yield "$100"
>>> hsbc = Bank() # when everything's ok the ATM gives you as much as you want
>>> corner_street_atm = hsbc.create_atm()
>>> print(corner_street_atm.next())
$100
>>> print(corner_street_atm.next())
$100
>>> print([corner_street_atm.next() for cash in range(5)])
['$100', '$100', '$100', '$100', '$100']
>>> hsbc.crisis = True # crisis is coming, no more money!
>>> print(corner_street_atm.next())
<type 'exceptions.StopIteration'>
>>> wall_street_atm = hsbc.create_atm() # it's even true for new ATMs
>>> print(wall_street_atm.next())
<type 'exceptions.StopIteration'>
>>> hsbc.crisis = False # trouble is, even post-crisis the ATM remains empty
>>> print(corner_street_atm.next())
<type 'exceptions.StopIteration'>
>>> brand_new_atm = hsbc.create_atm() # build a new one to get back in business
>>> for cash in brand_new_atm:
...     print cash
$100
$100
$100
$100
$100
$100
$100
$100
$100
...
```

It can be useful for various things like controlling access to a resource.

Itertools, your best friend

The itertools module contains special functions to manipulate iterables. Ever wish to duplicate a generator? Chain two generators? Group values in a nested list with a one liner? `Map / Zip` without creating another list?

Then just `import itertools` .

An example? Let's see the possible orders of arrival for a 4 horse race:

```
>>> horses = [1, 2, 3, 4]
>>> races = itertools.permutations(horses)
>>> print(races)
<itertools.permutations object at 0xb754f1dc>
>>> print(list(itertools.permutations(horses)))
[(1, 2, 3, 4),
 (1, 2, 4, 3),
 (1, 3, 2, 4),
 (1, 3, 4, 2),
 (1, 4, 2, 3),
 (1, 4, 3, 2),
 (2, 1, 3, 4),
 (2, 1, 4, 3),
 (2, 3, 1, 4),
 (2, 3, 4, 1),
 (2, 4, 1, 3),
 (2, 4, 3, 1),
 (3, 1, 2, 4),
 (3, 1, 4, 2),
 (3, 2, 1, 4),
 (3, 2, 4, 1),
 (3, 4, 1, 2),
 (3, 4, 2, 1),
 (4, 1, 2, 3),
 (4, 1, 3, 2),
 (4, 2, 1, 3),
 (4, 2, 3, 1),
 (4, 3, 1, 2),
 (4, 3, 2, 1)]
```

Understanding the inner mechanisms of iteration

Iteration is a process implying iterables (implementing the `__iter__()` method) and iterators (implementing the `__next__()` method). Iterables are any objects you can get an iterator from. Iterators are objects that let you iterate on iterables.

More about it in this article about how does the for loop work.

You can also `send` values to generators. If no value is sent then `x` is `None`, otherwise `x` takes on the sent value. Here is some info: http://docs.python.org/whatsnew/2.5.html#pep-342-new-generator-features

```
>>> def whizbang():
        for i in range(10):
            x = yield i
            print 'got sent:', x


>>> i = whizbang()
>>> next(i)
0
>>> next(i)
got sent: None
1
>>> i.send("hi")
got sent: hi
2
```