

Code Review Stack Exchange is a question and answer site for peer programmer code reviews. It's 100% free, no registration required.

[Take the 2-minute tour](#) 

## Number-to-word converter

My code works, but the aesthetic of the code seems to need some real work. The logic is simple enough: create three lists of words that will be used throughout the conversion, and create three functions: one for sub-1000 conversion, another for 1000-and-up conversion, and another function for splitting and storing into a list numbers above 1000.

First the code:

```
# Create the Lists of word-equivalents from 1-19, then one for the tens group.  
# Finally, a List of the (for lack of a better word) "zero-groups".
```

```
ByOne = [  
    "zero",  
    "one",  
    "two",  
    "three",  
    "four",  
    "five",  
    "six",  
    "seven",  
    "eight",  
    "nine",  
    "ten",  
    "eleven",  
    "twelve",  
    "thirteen",  
    "fourteen",  
    "fifteen",  
    "sixteen",  
    "seventeen",  
    "eighteen",  
    "nineteen"  
]
```

```
ByTen = [  
    "zero",  
    "ten",  
    "twenty",  
    "thirty",  
    "forty",  
    "fifty",  
    "sixty",  
    "seventy",  
    "eighty",  
    "ninety"  
]
```

```
zGroup = [  
    "",  
    "thousand",  
    "million",  
    "billion",  
    "trillion",  
    "quadrillion",  
    "quintillion",  
    "sextillion",  
    "septillion",  
    "octillion",  
    "nonillion",  
    "decillion",  
    "undecillion",  
    "duodecillion",  
    "tredecillion",
```

```

"quattuordecillion",
"sexdecillion",
"septendecillion",
"octodecillion",
"novemdecillion",
"vigintillion"
]

strNum = raw_input("Please enter an integer:\n>> ")

# A recursive function to get the word equivalent for numbers under 1000.

def subThousand(inputNum):
    num = int(inputNum)
    if 0 <= num <= 19:
        return ByOne[num]
    elif 20 <= num <= 99:
        if inputNum[-1] == "0":
            return ByTen[int(inputNum[0])]
        else:
            return ByTen[int(inputNum[0])] + "-" + ByOne[int(inputNum[1])]
    elif 100 <= num <= 999:
        rem = num % 100
        dig = num / 100
        if rem == 0:
            return ByOne[dig] + " hundred"
        else:
            return ByOne[dig] + " hundred and " + subThousand(str(rem))

# A looping function to get the word equivalent for numbers above 1000
# by splitting a number by the thousands, storing them in a list, and
# calling subThousand on each of them, while appending the correct
# "zero-group".

def thousandUp(inputNum):
    num = int(inputNum)
    arrZero = splitByThousands(num)
    lenArr = len(arrZero) - 1
    resArr = []
    for z in arrZero[::-1]:
        wrd = subThousand(str(z)) + " "
        zap = zGroup[lenArr] + ", "
        if wrd == " ":
            break
        elif wrd == "zero ":
            wrd, zap = "", ""
        resArr.append(wrd + zap)
        lenArr -= 1
    res = "".join(resArr).strip()
    if res[-1] == ",": res = res[:-1]
    return res

# Function to return a list created from splitting a number above 1000.

def splitByThousands(inputNum):
    num = int(inputNum)
    arrThousands = []
    while num != 0:
        arrThousands.append(num % 1000)
        num /= 1000
    return arrThousands

### Last part is pretty much just the output.

intNum = int(strNum)

if intNum < 0:
    print "Minus",
    intNum *= -1
    strNum = strNum[1:]

```

```

if intNum < 1000:
    print subThousand(strNum)
else:
    print thousandUp(strNum)

```

### Sample run:

```

>>>
Please enter an integer:
>> 95505896639631893
ninety-five quadrillion, five hundred and five trillion, eight hundred and ninety-six
billion, six hundred and thirty-nine million, six hundred and thirty-one thousand, eight
hundred and ninety-three
>>>

```

### Issues:

Basically, the peeve I'm having are as follows:

My first two functions seems to be taking up too many lines. The first one, `subThousand`, seems to be this way because of the check made against the last digit of a number from 20 to 99. The logic I applied was, if it ends in `0`, use the `ByTen` list. If it doesn't combined `ByTen` and `ByOne`. While effective, I feel like it could use some real work since I think it qualifies as a DRY-violation. Even the final part checking for numbers from 100 to 999 seems to follow the same pattern. Alas, I've tried paring it down but I've honestly hit a roadblock on this one insofar as trying to come up with a creative and clean solution.

My second function, `thousandUp` is quite the disaster. I tried coming up with a looping function that creates a list of one to three-digit numbers, so that I can call `subThousand` on each of them from the front to the back (hence the `arrZero[::-1]`). At the same time, after converting each element in the list to a word, I concatenate it with the appropriate equivalent in the `zGroup` list, or the list of the "zero-groups". However, I personally can't find a safer and more precise way of landing on the "correct" spot in the `zGroup` list to start concatenating.

To get around this, I took the length of the `splitByThousands` array, adjusted for `0`-index, and used it to get the appropriate "zero-append" (`zap`). Before the loop ends, I subtract one from its current value so that it's adjusted accordingly.

In addition, as I'm attempting to make the output as clean as possible, I add a `" "` to the `wrd` variable so it doesn't concatenate poorly with the zero-append, as well as add a `", "` to the zero-append to separate it from the next zero-group. However, there will be instances that there are no values for some zero-groups. To avoid showing stuff like `zero million`, I added a check inside. This is the part that makes me die a little inside:

```

for z in arrZero[::-1]:
    wrd = subThousand(str(z)) + " "
    zap = zGroup[lenArr] + ", "
    if wrd == " ":
        break
    elif wrd == "zero ":
        wrd, zap = "", ""
    resArr.append(wrd + zap)
    lenArr -= 1

```

It works, but it just doesn't look good. Is it possible to do this in list-comprehension form or a better for-loop without turning it into more confusing mush?


I must admit as well that the last part of the code sucks a little. I've done a lot of `str-->int` conversions inside the functions, then I did one more *outside* of them. On top of that, my approach to negative numbers is *hackish* at best (`print "Minus"`).

python   converting

edited May 4 at 18:44

 Jamal ♦  
18.6k   5   65   145

asked Mar 7 at 21:12

 Nanashi  
133   4

I recently tried the same task, but took a different approach. I wonder how you think it compares. Yours looks more sophisticated, ive not quite got to grips with the logic yet, but i am going ot try  
[to.codereview.stackexchange.com/questions/46273/...](http://codereview.stackexchange.com/questions/46273/...) – Woody Pride Apr 4 at 14:06

## 2 Answers

Try to extract the different pieces of logic in functions to make things clearer. In your case, you have defined little functions but the place where you are using them is a bit in the middle of nowhere.

Here's what you could do :

```
def get_number_as_words(strNum):
    intNum = int(strNum)
    if intNum < 0:
        print "Minus", # I didn't see this at the beginning but I'll fix it at the end
        intNum *= -1
        strNum = strNum[1:]

    if intNum < 1000:
        return subThousand(strNum)
    else:
        return thousandUp(strNum)
```

Also, all these functions should be properly documented using `docstrings`.

If your code can potentially be used as a module, it is advised to put the part which actually does stuff in a function and guard the call to this function behind a `if __name__ == "__main__":` condition. In your case, here's what I have :

```
def main():
    strNum = '95505896639631893' #raw_input("Please enter an integer:\n>> ")
    expected='ninety-five quadrillion, five hundred and five trillion, eight hundred and
ninety-six billion, six hundred and thirty-nine million, six hundred and thirty-one
thousand, eight hundred and ninety-three '
    print(get_number_as_words(strNum))
    print(expected)
    assert(get_number_as_words(strNum)==expected)

if __name__ == "__main__":
    main()
```

As I am messing around with your code, I use `assert` to detect quickly if I break the test case you have provided. Also, it shows something that we might want to fix on the long run : the trailing whitespace.

**subThousand:** You can use `divmod` to divide integers and get the quotient and the remainder.

**subThousand:** In successive `if`, `else if`, `else if`, etc, you do not need to check the same condition multiple times. Also, you can use `assert` to check that your function is used on proper values (described in the documentation).

**subThousand:** This function (and it might be the case for the others as well but I have to progress one function at a time) should probably be fed an integer and not a string. Operation on numbers should be enough for pretty much everything.

**subThousand:** You can use implicit evaluation of integers as boolean : `0` is `False`, anything else is `True`.

**subThousand:** As explained in Janne Karila's comment, it might be worth returning an empty string when the input is 0.

After taking into account the comments (except the last), the function looks like :

```
def subThousand(n):
    assert(isinstance(n,(int, long)))
    assert(0 <= n <= 999)
    if n <= 19:
        return ByOne[n]
    elif n <= 99:
        q, r = divmod(n, 10)
```

```

    return ByTen[q] + ("-" + subThousand(r) if r else "")
else:
    q, r = divmod(n, 100)
    return ByOne[q] + " hundred" + (" and " + subThousand(r) if r else "")

```

I am a big fan of the ternary operator but it's purely personal. Also I used a recursive call to make things more consistent.

**splitByThousands:** By taking into account previous comments (types, divmode, etc), you can already get something more concise :

```

def splitByThousands(n):
    assert(isinstance(n,(int, long)))
    assert(0 <= n)
    res = []
    while n:
        n, r = divmod(n, 1000)
        res.append(r)
    return res

```

**thousandUp:** The same comments are still relevant.

**thousandUp:** We maintain one list `resArr` and a number `lenArr` through the function but we can see that the property `lenArr + len(resArr) + 1 == len(arrZero)` is (almost) always true. One can easily get convinced of this or add `assert(lenArr + len(resArr) + 1 == len(arrZero))` everywhere. Thus `lenArr = len(arrZero) - len(resArr) - 1`.

**thousandUp:** The block

```

wrd = subThousand(z) + " "
zap = zGroup[len(arrZero) - len(resArr) - 1] + ", "
if wrd == " ":
    break
elif wrd == "zero ":
    wrd, zap = "", ""
resArr.append(wrd + zap)

```

can be simplified a lot :

1. there is not point in adding a whitespace to `wrd` at this stage.
2. you do not need to use `zGroup` in all cases.
3. as `subThousand(X)` is "zero" only if `X==0`, `wrd == "zero"` becomes `z == 0`
4. as `subThousand(X)` is never an empty string, this test has no reason to be there. (If you ever perform the change described above, `subThousand(X)` will be empty when `X==0` which is a case already handled).

Thus the block becomes :

```

if z:
    wrd = subThousand(z)
    zap = zGroup[len(arrZero) - len(resArr) - 1] + ", "
else:
    wrd, zap = "", ""
resArr.append(wrd + " " + zap)

```

which can then be rewritten :

```

if z:
    wrd_zap = subThousand(z) + " " + zGroup[len(arrZero) - len(resArr) - 1] + ", "
else:
    wrd_zap = ""
resArr.append(wrd_zap)

```

Then, because there is not point in adding spaces because all elements of the array already end with a space, we can do :

```

if z:

```

```

    wrd_zap = subThousand(z) + " " + zGroup[len(arrZero) - len(resArr) - 1] + ", "
else:
    wrd_zap = ""
resArr.append(wrd_zap)

```

**thousandUp:** Once the code is simplified, we realise that as described in Janne Karila's comment, we could use `", "` to call `join()`, killing many birds with one stone :

- the hack to remove the trailing comma is not required anymore
- the trailing whitespace also disappears (this is a nice side-effect as the `strip()` now performs what is was supposed to do in the first place but was then prevented by the commas)
- everything becomes easier

We now have :

```

for z in arrZero[::-1]:
    resArr.append(subThousand(z) + " " + zGroup[len(arrZero) - len(resArr) - 1] if z else "")
return ", ".join(resArr).strip()

```

**thousandUp:** Something that wasn't obvious at first but starts to be : for each element in `arrZero`, we add an element in `resArr` so that `len(arrZero) - len(resArr) - 1` goes from `len(arrZero)-1` to `0`. It looks like we could achieve this with `enumerate()`. We could do it in a complicated way but let's keep things simple instead : let's iterate in the obvious order and call `reversed` at the very end when joining.

```

resArr = []
for i,z in enumerate(arrZero):
    resArr.append(subThousand(z) + " " + zGroup[i] if z else "")
return ", ".join(reversed(resArr))

```

But he, what do we have here ?! A typical list comprehension

```

def thousandUp(n):
    assert(isinstance(n,(int, long)))
    return ", ".join(reversed([subThousand(z) + " " + zGroup[i] if z else "" for i,z in
enumerate(splitByThousands(n))]))

```

*Oops, I might have gone too far*. Actually, we can go even further to handle the trailing whitespaces in a smart way :

```

def thousandUp(n):
    assert(isinstance(n,(int, long)))
    assert(0 <= n)
    return ", ".join(reversed([subThousand(z) + (" " + zGroup[i] if i else "") if z else ""
for i,z in enumerate(splitByThousands(n))]))

```

I have kept this mistake for too long to fix the whole answer but :

```

def thousandUp(n):
    assert(isinstance(n,(int, long)))
    assert(0 <= n)
    return ", ".join(reversed([subThousand(z) + (" " + zGroup[i] if i else "") for i,z in
enumerate(splitByThousands(n)) if z]))

```

is probably a better version (we don't get repetitions of the separator).

**get\_number\_as\_words:** Finally, this should also take an integer as a parameter. Also, it is probably the right place to handle 0 as a special value. Moreover, `thousandUp()` handles properly inputs smaller than 1000.

At the end, here's what my code is like :

```

def subThousand(n):
    assert(isinstance(n,(int, long)))
    assert(0 <= n <= 999)
    if n <= 19:

```

```

    return ByOne[n]
elif n <= 99:
    q, r = divmod(n, 10)
    return ByTen[q] + ("-" + subThousand(r) if r else "")
else:
    q, r = divmod(n, 100)
    return ByOne[q] + " hundred" + (" and " + subThousand(r) if r else "")

def thousandUp(n):
    assert(isinstance(n,(int, long)))
    assert(0 <= n)
    return ", ".join(reversed([subThousand(z) + (" " + zGroup[i] if i else "") if z else "" for i,z in enumerate(splitByThousands(n))]))

def splitByThousands(n):
    assert(isinstance(n,(int, long)))
    assert(0 <= n)
    res = []
    while n:
        n, r = divmod(n, 1000)
        res.append(r)
    return res

def get_number_as_words(n):
    assert(isinstance(n,(int, long)))
    if n==0:
        return "Zero"
    return ("Minus " if n < 0 else "") + thousandUp(abs(n))

def main():
    n = 95505896639631893 # int(raw_input("Please enter an integer:\n>> "))
    expected='ninety-five quadrillion, five hundred and five trillion, eight hundred and
ninety-six billion, six hundred and thirty-nine million, six hundred and thirty-one
thousand, eight hundred and ninety-three'
    print(get_number_as_words(n))
    print(expected)
    assert(get_number_as_words(n)==expected)

if __name__ == "__main__":
    main()

```

I hope you will find it more pleasing and that my comments will help you. Also, despite the big number of comments, your initial code was pretty good.

*Disclaimer : I have done everything only based on your example (which was enough to spot mistakes as I was writing them). I might have missed a few things or done a few things wrong so I have tried to detail the different steps as much as possible just in case.*

**Edit** I just thought that you could change `splitByThousands` to make it `yield` values instead of returning a list...

edited Mar 8 at 2:38

answered Mar 8 at 2:15



Josay

10.6k 11 41

+1: Marking and accepting this as answer. The sheer amount of standards and practices displayed here is invaluable. I will read this forensics work slowly and will be documenting part-by-part responses in my original post, all while improving my code. Also, suddenly, list comprehension looks a *little* too intimidating. I'll be weighing whether I go for readability or for conformity on that end. Off the top, though, thanks for the docstrings suggestion. I'm reading up on it now. Very excited to apply the practices in your post soon! Thanks a lot, Josay. :) — Nanashi Mar 8 at 14:08

A couple of thoughts:

- You only want to output `zero` in the special case when the number is 0. It would be easiest to handle that special case at the outermost level.
- Use `"` and `".join` and `", ".join` to construct strings from parts. The delimiter will be automatically left out when there are less than two parts.

Here's how I would rewrite `subThousand` as two functions:

```
def subThousand(inputNum):
    """Convert a number < 1000 to words, and 0 to the empty string
    """
    num = int(inputNum) #this should be done at higher Level
    hundreds, ones = divmod(num, 100)

    parts = []
    if hundreds:
        parts.append(ByOne[hundreds] + " hundred")
    if ones:
        parts.append(subHundred(ones))

    return " and ".join(parts)

def subHundred(num):
    """Convert a number < 100 to words, and 0 to the empty string
    """
    if num >= 20:
        tens, ones = divmod(num, 10)
    else:
        tens, ones = 0, num

    parts = []
    if tens:
        parts.append(ByTen[tens])
    if ones:
        parts.append(ByOne[ones])

    return "-".join(parts)
```

edited Mar 8 at 11:03

answered Mar 7 at 21:54



Janne Karila  
4,230 5 18

---

+1: Amazing modification! I think missing `divmod` and failing to realize it exists within Python seems to be a glaring issue. In C, I pretty much use the logic I used in my original code, hence carrying it over here as well. Also, love the use of the `" and ".join` -- it's so smart I tear up looking at it right now. Thanks a lot! — Nanashi Mar 8 at 14:04

---