1. Gematria: Numeric encoding of text using ASCII values

Gematria is a system for assigning a number to a word by summing the numeric values of each of the characters [2]. In the standard encoding (*Mispar hechrechi*), each characters of the Hebrew alphabet is assigned a numeric values ranging from 1 to 400, but there are more than a dozen other methods for calculating the numeric value for the letters. To encode a word, these values are added together. Revelation 13:18 from the Christian Bible says "Let the one who has insight calculate the number of the wild beast, for it is a man’s number, and its number is 666." Some scholars believe that number is derived from the encoding of the characters representing Nero Caesar’s name and title and was used as a way of writing about the Roman emperor without naming him.

We will write a program called `gematria.py` that will numerically encode each word in a given text by similarly adding numeric values for the characters in each word. There are many ways we could assign these values; for instance, we could start by given "a" the value `1`, "b" the value `2`, and so forth. Instead, we will use the ASCII table [3] to derive a numeric for English alphabet characters. For non-English characters, we could consider using a Unicode value, but this exercise will stick to ASCII letters.

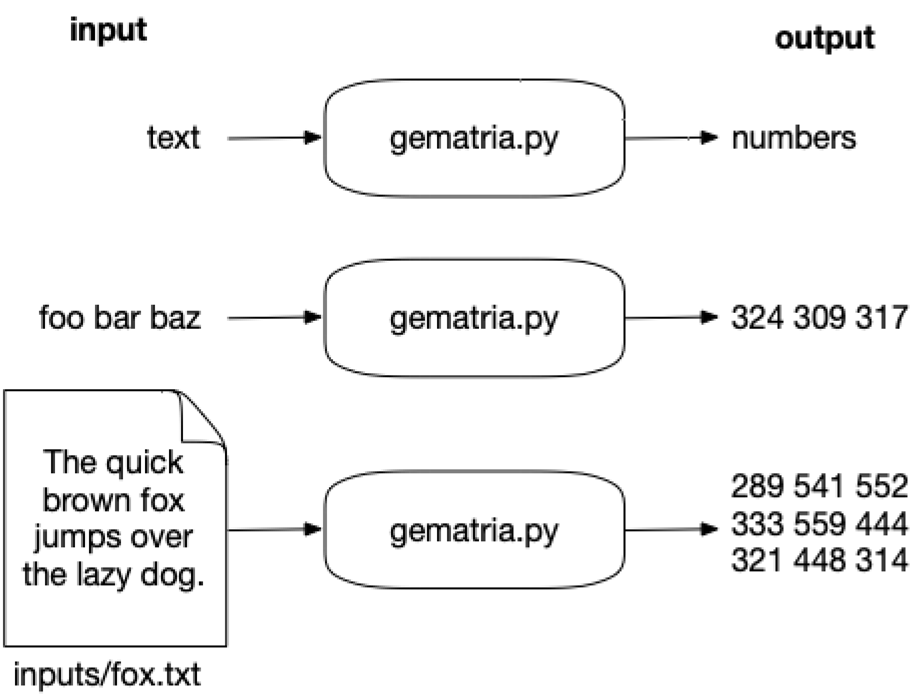
The input text may be given on the command line:

```
$ ./gematria.py 'foo bar baz'
324 309 317
```

Or in a file:

```
$ ./gematria.py ../inputs/fox.txt
289 541 552 333 559 444 321 448 314
```

Here is a string diagram showing how the program should work:



In this exercise, you will:

- Learn about the `ord()` and `chr()` functions
- Explore how characters are organized in the ASCII table
- Understand character ranges used in regular expression
- Use the `re.sub()` function
- Learn how `map()` can be written without `lambda`
- Use the `sum()` function and see how that relates to using `reduce()`
- Learn how to perform case-insensitive string sorting

1.1. Writing `gematria.py`

I will always recommend you start your programs in some way that avoids having to type all the boilerplate text. Either copy `template/template.py` to `18_gematria/gematria.py` or use `new.py gematria.py` in the `18_gematria` directory to create a starting point. Modify the program until it prints the following usage if given no arguments or the `-h` or `--help` flag:

```
$ ./gematria.py -h
usage: gematria.py [-h] text

Gematria

positional arguments:
  text        Input text or file

optional arguments:
  -h, --help  show this help message and exit
```

As in previous exercises, the input may come from the command line or from a file. I suggest you copy the code you used in "Howler" to handle this, then modify your `main()` to the following:

```
def main():
    args = get_args()
    print(args.text)
```

Verify that your program will print text from the command line:

```
$ ./gematria.py 'Death smiles at us all, but all a man can do is smile back.'
Death smiles at us all, but all a man can do is smile back.
```

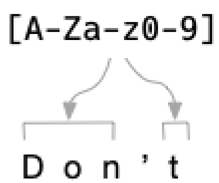
Or from a file:

```
$ ./gematria.py ../inputs/spiders.txt
Don't worry, spiders,
I keep house
casually.
```

1.1.1. Cleaning a word

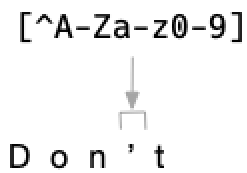
Let’s discuss how a single word will be encoded as it will affect how we will break the text in the next section. In order to be absolutely sure we are only dealing with ASCII values, let’s remove anything that is not an upper- or lowercase English alphabet character or any of the Arabic numerals 0-9. We can define that class of characters using the regular expression `[A-Za-z0-9]`. We can use the `re.findall()` function we used in "Mad Libs" to find all the characters in `word` that match this class. In the word "Don’t," we should expect to find everything except the apostrophe:

```
>>> re.findall('[A-Za-z0-9]', "Don't")
['D', 'o', 'n', 't']
```

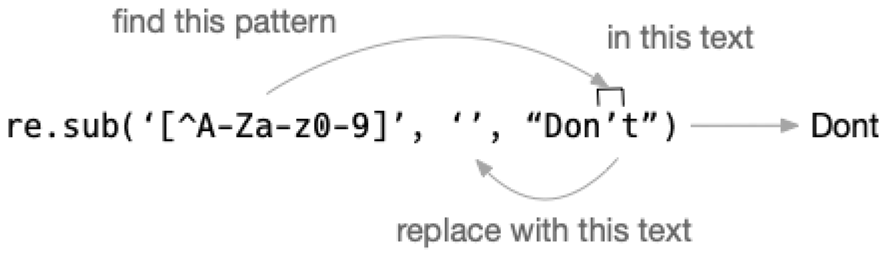


Anything *not* in that class can be defined by placing a caret (`^` or "hat") as the first character inside the class like `[^A-Za-z0-9]`. Now we would expect to match *only* the apostrophe:

```
>>> import re
>>> re.findall('[^A-Za-z0-9]', "Don't")
["'"]
```



We can use the `re.sub()` function to replace any characters in that second class with the empty string. As you learned in "Mad Libs," this will replace *all* occurrences of the pattern unless we use the `count=n` option:




```
>>> word = re.sub('[^A-Za-z0-9]', '', "Don't")
>>> word
'Dont'
```

We will want use this operation to clean each word that we'll encode.

1.1.2. Ordinal character values and ranges

We will encode a string like "Dont" by converting *each character* to a numeric value and then adding them together, so let's first figure out how to encode a single character. Python has a function called `ord()` that will convert a character to its "ordinal" value (its order in the ASCII table):

```
>>> ord('D')
68
>>> ord('o')
111
```

The `chr()` function works in reverse to convert a number to a character:

```
>>> chr(68)
'D'
>>> chr(111)
'o'
```

Here is a table showing the lower-order ASCII values 0-127 [4]. For simplicity's sake, I show "NA" for "not available" for the ones up to index 31 as they are not printable:

```
$ ./asciitbl.py
  0 NA      16 NA      32 SPACE   48 0       64 @       80 P       96 `      112 p
  1 NA      17 NA      33 !       49 1       65 A       81 Q       97 a      113 q
  2 NA      18 NA      34 "       50 2       66 B       82 R       98 b      114 r
  3 NA      19 NA      35 #       51 3       67 C       83 S       99 c      115 s
  4 NA      20 NA      36 $       52 4       68 D       84 T      100 d      116 t
  5 NA      21 NA      37 %       53 5       69 E       85 U      101 e      117 u
  6 NA      22 NA      38 &       54 6       70 F       86 V      102 f      118 v
  7 NA      23 NA      39 '       55 7       71 G       87 W      103 g      119 w
  8 NA      24 NA      40 (       56 8       72 H       88 X      104 h      120 x
  9 NA      25 NA      41 )       57 9       73 I       89 Y      105 i      121 y
 10 NA      26 NA      42 *       58 :       74 J       90 Z      106 j      122 z
 11 NA      27 NA      43 +       59 ;       75 K       91 [      107 k      123 {
 12 NA      28 NA      44 ,       60 <       76 L       92 \      108 l      124 |
 13 NA      29 NA      45 -       61 =       77 M       93 ]      109 m      125 }
 14 NA      30 NA      46 .       62 >       78 N       94 ^      110 n      126 ~
 15 NA      31 NA      47 /       63 ?       79 O       95 _      111 o      127 DEL
```

We can use a `for` loop to cycle through all the characters in a string:

```
>>> word = "Dont"
>>> for char in word:
...     print(char, ord(char))
...
D 68
o 111
n 110
t 116
```

Note that upper- and lowercase letters have different `ord()` values. It makes sense because they are two different letters:

```
>>> ord('D')
68
>>> ord('d')
100
```

We can iterate over the values from "a" to "z" by finding their `ord()` values:

```
>>> [chr(n) for n in range(ord('a'), ord('z') + 1)]
['a', 'b', 'c', 'd', 'e', 'f', 'g', 'h', 'i', 'j', 'k', 'l', 'm',
 'n', 'o', 'p', 'q', 'r', 's', 't', 'u', 'v', 'w', 'x', 'y', 'z']
```

The letters "a" through "z" lies contiguously in the ASCII table. You can do the same for "A" to "Z" and "0" to "9" which is why we can use `[A-Za-z0-9]` as a regex.

Note that the uppercase letters have lower ordinal values than their lowercase versions, which is why you cannot use the range `[a-Z]`. Try this in the REPL and note the error you get:

```
>>> re.findall('[a-Z]', word)
```

The last line I see is this:

```
re.error: bad character range a-Z at position 1
```

You *can* use the range `[A-z]`:

```
>>> re.findall('[A-z]', word)
['D', 'o', 'n', 't']
```

But see that that "Z" and "a" are not contiguous:

```
>>> ord('Z'), ord('a')
(90, 97)
```

There are other characters in between them:

```
>>> [chr(n) for n in range(ord('Z') + 1, ord('a'))]
['[', '\\', ']', '^', '_', '`']
```

If we try to use that range on all the printable characters, you'll see that it matches characters that are not letters:

```
>>> import string
>>> re.findall('[A-z]', string.printable)
['a', 'b', 'c', 'd', 'e', 'f', 'g', 'h', 'i', 'j', 'k', 'l', 'm',
 'n', 'o', 'p', 'q', 'r', 's', 't', 'u', 'v', 'w', 'x', 'y', 'z',
 'A', 'B', 'C', 'D', 'E', 'F', 'G', 'H', 'I', 'J', 'K', 'L', 'M',
 'N', 'O', 'P', 'Q', 'R', 'S', 'T', 'U', 'V', 'W', 'X', 'Y', 'Z',
 '[', '\\', ']', '^', '_', '`']
```

That is why it is safest to specify the characters we want as the three ranges, [A-Za-z0-9], which you may sometimes hear pronounced as "A to Z, a to z, zero to nine" as it assumes you understand that there are two "a to z" ranges which are distinct according to their case.

1.1.3. Summing and reducing

Let's keep reminding ourselves what the goal is here: convert all the characters in a word and them sum those values. There is a handy Python function called sum() that will add a list of numbers:

```
>>> sum([1, 2, 3])
6
```

We can manually encode the string "Dont" by calling ord() on each letter and passing the results as a list to sum():

```
>>> sum([ord('D'), ord('o'), ord('n'), ord('t')])
405
```

So the question is how to apply the function ord() to all the elements of a str and give the result to sum(). You've seen this pattern many times now. What's the first tool you'll reach for? We can always start with our handy for loop:

```
>>> word = 'Dont'
>>> vals = []
>>> for char in word:
...     vals.append(ord(char))
...
>>> vals
[68, 111, 110, 116]
```

Can you see how to make that into a single line using a list comprehension?

```
>>> vals = [ord(char) for char in word]
>>> vals
[68, 111, 110, 116]
```

And from there, we can move to a map():

```
>>> vals = map(lambda char: ord(char), word)
>>> list(vals)
[68, 111, 110, 116]
```

Here I'd like to show that the `map()` version here doesn't even need the `lambda` declaration because the `ord()` function expects a single value which is exactly what it will get from `map()`. Here is a nicer way to write it:

```
>>> vals = map(ord, word)
>>> list(vals)
[68, 111, 110, 116]
```

To my eye, that is a really beautiful piece of code! Now we can `sum()` that to get a final value for our `word`:

```
>>> sum(map(ord, word))
405
```

Which is correct:

```
>>> sum([68, 111, 110, 116])
405
```

Using `functools.reduce()`

If Python has a `sum()` function, you might suspect it also has a `product()` function to multiply a list of numbers together. Alas, this is not a built-in function, but it does represent a common idea of *reducing* a list of values into a single value. The `reduce()` function from the `functools` module provides a generic way to reduce a list. Let's consult the documentation for how to use it:

```
>>> from functools import reduce
>>> help(reduce)
reduce(...)
    reduce(function, sequence[, initial]) -> value

    Apply a function of two arguments cumulatively to the items of a sequence,
    from left to right, so as to reduce the sequence to a single value.
    For example, reduce(lambda x, y: x+y, [1, 2, 3, 4, 5]) calculates
    ((((1+2)+3)+4)+5).  If initial is present, it is placed before the items
    of the sequence in the calculation, and serves as a default when the
    sequence is empty.
```

This is another higher-order function which wants *another function* as the first argument, just like `map()` and `filter()`. The documentation shows us how to write our own `sum()` function:

```
>>> reduce(lambda x, y: x + y, [1, 2, 3, 4, 5])
15
```

If we change the `+` operator to `*`, then we have a product:

```
>>> reduce(lambda x, y: x * y, [1, 2, 3, 4, 5])
120
```

Here is how you might write a function for this:

```python
def product(vals):
    return reduce(lambda x, y: x * y, vals)
```

And now you can call it:

```
>>> product(range(1,6))
120
```

Instead of writing our own `lambda`, we can use *any function that expects two arguments*. The `operator.mul` function fits this bill:

```
>>> import operator
>>> help(operator.mul)
mul(a, b, /)
    Same as a * b.
```

So it would be easier to write this:

```
def product(vals):
    return reduce(operator.mul, vals)
```

Fortunately, the `math` module also contains a `prod()` function you can use:

```
>>> import math
>>> math.prod(range(1,6))
120
```

If you think about it, the `str.join()` method also reduces a `list` of strings to a single `str` value. Here's how we can write our own:

```
def join(sep, vals):
    return reduce(lambda x, y: x + sep + y, vals)
```

I much prefer the syntax of calling this `join` over the `str.join()` function:

```
>>> join(', ', ['Hey', 'Nonny', 'Nonny'])
'Hey, Nonny, Nonny'
```

Whenever you have a `list` of any values that you want to combine to produce a single value, consider using the `reduce()` function!

1.1.4. Encoding the words

That was a lot just to get to summing the ordinal values of the characters, but wasn't it fascinating to explore? Let's get back on track, though! We can create a function to encapsulate the idea of converting a word into a numeric value derived from summing the ordinal values of the characters. I called mine `word2num()`, and here is my test:

```
def test_word2num():
    """Test word2num"""
    assert word2num("a") == "97"
    assert word2num("abc") == "294"
    assert word2num("ab'c") == "294"
    assert word2num("4a-b'c,") == "346"
```

Notice that my function returns a `str` value, not an `int`. This is because I want to use the result with the `str.join()` function that only accepts `str` values. So `'405'` instead of `405`:

```
>>> from gematria import word2num
>>> word2num("Don't")
'405'
```

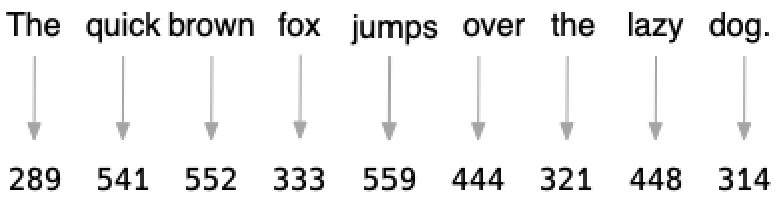
To summarize, the `word2num()` function accepts a word, removes unwanted characters, converts the remaining characters to `ord()` values, and returns a `str` representation of the `sum()` of those values.

1.1.5. Breaking the text

The tests expect you to maintain the same line breaks as the original text, so I recommend you use `str.splitlines()` as in other exercises. In "Friar" and "Scrambler," we used different regexes to split each line into "words," a process sometimes called "tokenization" in programs that deal with Natural Language Processing (NLP). If you write a `word2num()` that passes the tests I provide, then you can use `str.split()` to break a line on spaces because the function will ignore anything that is not a character or number. You are, of course, welcome to break the line into words using whatever means you like.

This code will maintain the line breaks and reconstruct the text. Can you modify it to add the `word2num()` function so that it instead prints out encoded words?

```
def main():
    args = get_args()
    for line in args.text.splitlines():
        for word in line.split():
            # what goes here?
            print(' '.join(line.split()))
```



The output will be one number for each word:

```
$ ./gematria.py ../inputs/fox.txt
289 541 552 333 559 444 321 448 314
```

Time to finish writing the solution. Be sure to use the tests! See you on the flip side.

1.2. Solution

```
#!/usr/bin/env python3
"""Gematria"""

import argparse
import os
import re


# --------------------------------------------------
def get_args():
    """Get command-line arguments"""

    parser = argparse.ArgumentParser(
        description='Gematria',
        formatter_class=argparse.ArgumentDefaultsHelpFormatter)

    parser.add_argument('text', metavar='text', help='Input text or file') ①

    args = parser.parse_args()                    ②

    if os.path.isfile(args.text):                 ③
        args.text = open(args.text).read().rstrip()  ④

    return args                                   ⑤


# --------------------------------------------------
def main():
    """Make a jazz noise here"""

    args = get_args()                             ⑥

    for line in args.text.splitlines():           ⑦
        print(' '.join(map(word2num, line.split()))) ⑧


# --------------------------------------------------
def word2num(word): ⑨
    """Sum the ordinal values of all the characters"""

    return str(sum(map(ord, re.sub('[^A-Za-z0-9]', '', word)))) ⑩


# --------------------------------------------------
def test_word2num(): ⑪
    """Test word2num"""

    assert word2num("a") == "97"
```

```
    assert word2num("abc") == "294"
    assert word2num("ab'c") == "294"
    assert word2num("4a-b'c,") == "346"


# --------------------------------------------------
if __name__ == '__main__':
    main()
```

① The `text` argument is a string which might be a file name.

② Get the parsed command-line arguments.

③ Check if the `text` argument names an existing file.

④ If it does, overwrite the `args.text` with the contents of the file.

⑤ Return the fixed up arguments.

⑥ Get the parsed arguments.

⑦ Split `args.text` on newlines to retain line breaks.

⑧ Split the `line` on spaces, `map()` the result through `word2num()`, then join that result on spaces.

⑨ Define a function to convert a word to a number.

⑩ Use `re.sub()` to remove anything not an alpha-numeric character, `map()` the resulting string through the `ord()` function, `sum()` the ordinal values of the characters, and return a `str` representation of the number.

⑪ Define a function to test the `word2num()` function.

1.3. Discussion

I trust you understand the `get_args()` as we've used this exact code several times now. Let's jump to the `word2num()` function.

1.3.1. Writing `word2num()`

I could have written the function like this:

```
def word2num(word):
    vals = []                                   ①
    for char in re.sub('[^A-Za-z0-9]', '', word): ②
        vals.append(ord(char))                  ③

    return str(sum(vals))                       ④
```

① Initialize an empty `list` to hold the values.

② Iterate all the characters returned from `re.sub()`.

③ Convert the character to an ordinal value and append that to the values.

④ Sum the values and return a string representation.

That’s four lines of code instead of one I wrote. I would at least rather use a list comprehension which collapses three lines of code into one:

```
def word2num(word):
    vals = [ord(char) for char in re.sub('[^A-Za-z0-9]', '', word)]
    return str(sum(vals))
```

Which can be written in one line though it could be argued that readability suffers:

```
def word2num(word):
    return str(sum([ord(char) for char in re.sub('[^A-Za-z0-9]', '', word)]))
```

I still think the `map()` version is the most readable and concise:

```
def word2num(word):
    return str(sum(map(ord, re.sub('[^A-Za-z0-9]', '', word))))
```

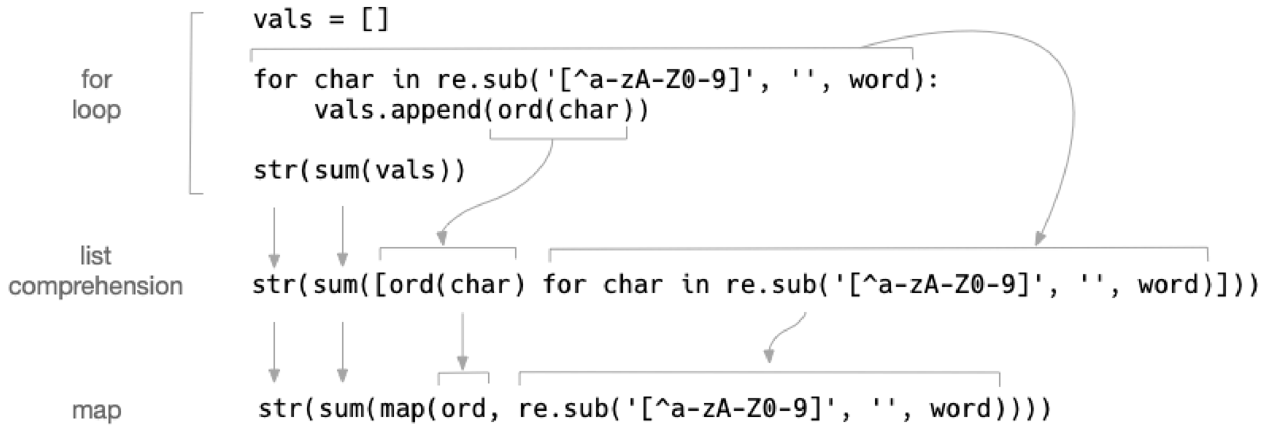
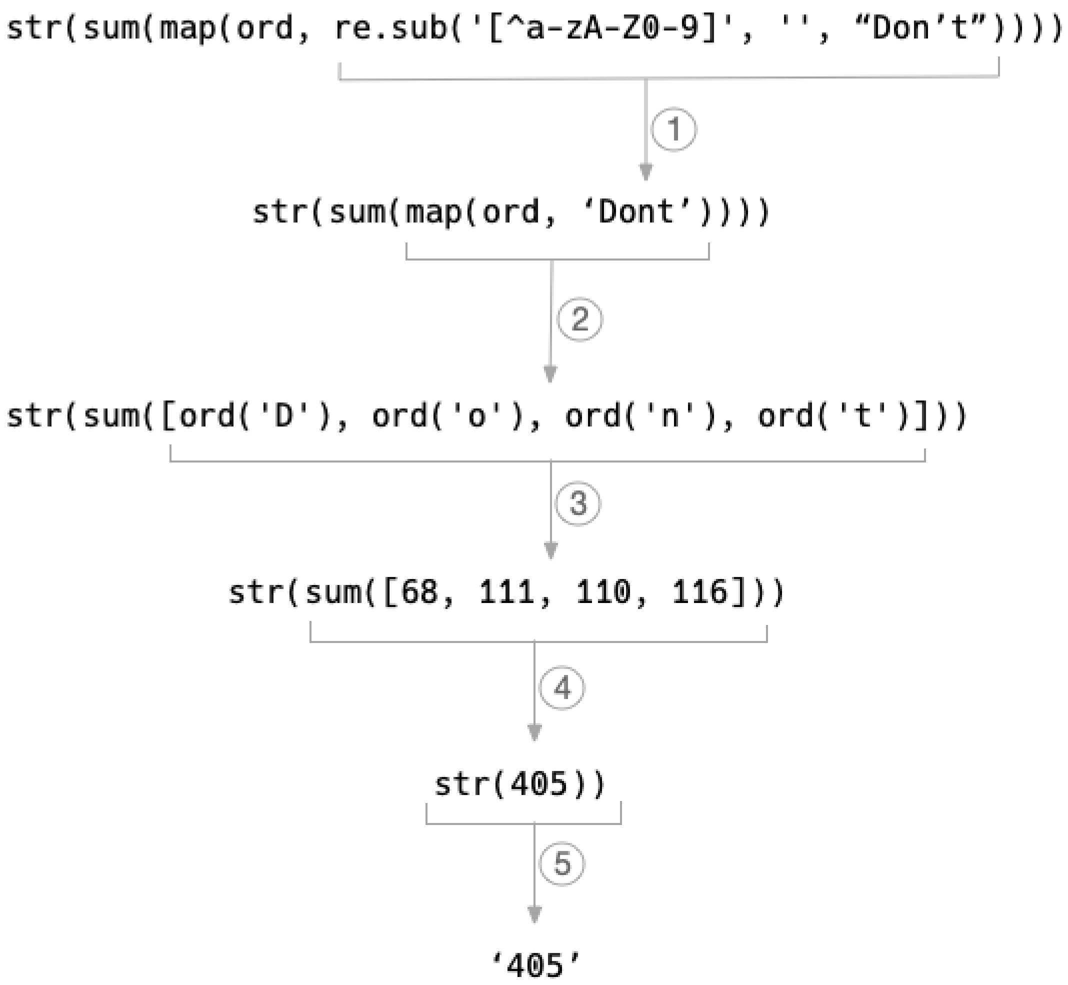


Figure 18. 1. How the `for` loop, a list comprehension, and a `map()` relate to each other.

Here’s a diagram to help you see how the data moves through the `map()` version with the string "Don’t":



1. The `re.sub()` function will replace any character not in the character class with the empty string. This will turn a word like "Don’t" into "Dont" (without the apostrophe).
2. The `map()` will apply the given function `ord()` to each element of a sequence. Here that "sequence" is a word, so it will use each character of the word.
3. The result of `map()` is a new `list` where each character from "Dont" is given to the `ord()` function.
4. The results of the calls to `ord()` will be a `list` of `int` values, one for each letter.
5. The `sum()` function will reduce a list of numbers to a single value by adding them together.
6. We will be using the `str.join()` function to create the output, and that function will throw an exception if all the values are not strings, so we use the `str()` function to turn the return from `sum()` into a string representation of the number.

1.3.2. Sorting

The point of this exercise was less about the `ord()` and `chr()` functions and more about exploring regular expressions, function application, and how characters are represented inside programming languages like Python. For instance, sorting of strings is case-sensitive. Note that all the uppercase letters get sorted as a group and then the lowercase:

```
>>> words = 'banana Apple Cherry anchovies cabbage Beets'
>>> sorted(words)
['Apple', 'Beets', 'Cherry', 'anchovies', 'banana', 'cabbage']
```

This is because all the uppercase ordinal values are lower than those of the lowercase letters. In order to perform a case-insensitive sorting of strings, you can use `key=str.casefold`. The `str.casefold()` function will return "a version of the string suitable for caseless comparisons". We are using here *without parentheses* because we are passing *the function itself* as the argument for `key`:

```
>>> sorted(words, key=str.casefold)
['anchovies', 'Apple', 'banana', 'Beets', 'cabbage', 'Cherry']
```

If you add the parens, it will cause an exception. This is exactly the same as how we pass functions as arguments to `map()` and `filter()`:

```
>>> sorted(words, key=str.casefold())
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
TypeError: descriptor 'casefold' of 'str' object needs an argument
```

The option is the same with `list.sort()` if you prefer to sort the list in-place:

```
>>> words.sort(key=str.casefold)
>>> words
['anchovies', 'Apple', 'banana', 'Beets', 'cabbage', 'Cherry']
```

Unix command-line tools like the `sort` program behave in the same way due to the same representation of characters. Given a file of these same words:

```
$ cat words.txt
banana
Apple
Cherry
anchovies
cabbage
Beets
```

The `sort` program on my Mac [5] will first sort the uppercase words and then the lowercase:

```
$ sort words
Apple
Beets
Cherry
anchovies
banana
cabbage
```

I have to read the `sort` manual page (via `man sort`) to find the `-f` flag to perform a case-insensitive sort:

```
$ sort -f words
anchovies
Apple
banana
Beets
cabbage
Cherry
```

1.3.3. Testing

I would like to take a moment to point out how often I use my own tests. Every time I write an alternate version of a function or program, I run my own tests to verify that I'm not accidentally showing you buggy code. Having a test suite gives me the freedom and confidence to extensively refactor my programs because I know I can check my work. If I ever find a bug in my code, I add a test to verify that the bug exists. Then I fix the bug and verify that it's handled. I know if I accidentally reintroduce that bug, my tests will catch it!

For the purposes of this book, I try to never write a program over 100 lines. It's common for programs to grow to thousands of lines of code (LOC) spread over dozens of modules. I recommend you start writing and using tests no matter how small you start. It's a good habit to establish early on and will only help you as you write more LOC!

1.4. Review

- The `ord()` function will return the ordinal value of a character which is its position in the ASCII table.
- The `chr()` function will return the character at a given position in the ASCII table.
- We can use character ranges like `a-z` in regular expressions because the characters lie contiguously in the ASCII table.
- The `re.sub()` function will replace matching patterns of text in a string with new values such as replacing all non-characters with the empty string to remove punctuation and whitespace.
- A `map()` can be written with a direct function reference instead of a `lambda` if the function expects a single positional argument.

- The `sum()` function reduces a list of numbers using addition. We can manually write a version of this using the `functools.reduce()` function. Considers of how that relates to the "map-reduce" concept we discussed before.
- To perform a case-insensitive sort of string values, use `key=str.casefold` option with both the `sorted()` and `list.sort()` functions.

1.5. Going Further

- Create a version that will handle non-English characters by using Unicode values for characters. You'll find that the idea of a "character" is quite different in Unicode. Something that appears to be one entity may actually be composed of more than one Unicode character, hence ideas like "grapheme clusters" [6] and "code points" [7].
- Analyze text files to find other words that sum to the value 666. Are these particularly scary words?

[3] https://en.wikipedia.org/wiki/ASCII

[4] I included the `asciitbl.py` program I used to create this.

[5] The GNU coreutils 8.30 version on one of my Linux machines will perform a case-insensitive sort by default. How does your `sort` work?

[6] https://unicode.org/reports/tr29/

[7] https://en.wikipedia.org/wiki/Code_point