The previous task provided a way to compute three elements i, j, k of S whose sum is zero—if there exist three such elements. Suppose you wanted to determine if there were a hundred elements of S whose sum is zero. What would go wrong if you used the approach used in the previous task? Can you think of a clever way to quickly and reliably solve the problem, even if the integers making up S are very large? (If so, see me immediately to collect your Ph.D.)

Obtaining a list or set from another collection

Python can compute a set from another collection (e.g. a list) using the constructor `set(·)`. Similarly, the constructor `list(·)` computes a list, and the constructor `tuple(·)` computes a tuple

```
>>> set([0,1,2,3,4,5,6,7,8,9])
{0, 1, 2, 3, 4, 5, 6, 7, 8, 9}
>>> set([1,2,3])
{1, 2, 3}
>>> list({1,2,3})
[1, 2, 3]
>>> set((1,2,3))
{1, 2, 3}
```

Task 0.5.17: Find an example of a list `L` such that `len(L)` and `len(list(set(L)))` are different.

0.5.7 Other things to iterate over

Tuple comprehensions—not! Generators

One would expect to be able to create a tuple using the usual comprehension syntax, e.g. `(i for i in [1,2,3])` but the value of this expression is not a tuple. It is a *generator*. Generators are a very powerful feature of Python but we don't study them here. Note, however, that one can write a comprehension over a generator instead of over a list or set or tuple. Alternatively, one can use `set(·)` or `list(·)` or `tuple(·)` to transform a generator into a set or list or tuple.

Ranges

A range plays the role of a list consisting of the elements of an arithmetic progression. For any integer n, `range(n)` represents the sequence of integers from 0 through $n-1$. For example, `range(10)` represents the integers from 0 through 9. Therefore, the value of the following comprehension is the sum of the squares of these integers: `sum({i*i for i in range(10)})`.

Even though a range represents a sequence, it is not a list. Generally we will either iterate through the elements of the range or use `set(·)` or `list(·)` to turn the range into a set or list.

```
>>> list(range(10))
[0, 1, 2, 3, 4, 5, 6, 7, 8, 9]
```

Task 0.5.18: Write a comprehension over a range of the form `range(n)` such that the value of the comprehension is the set of odd numbers from 1 to 99.

You can form a range with one, two, or three arguments. The expression `range(a,b)` represents the sequence of integers $a, a+1, a+2, \ldots, b-1$. The expression `range(a,b,c)` represents $a, a+c, a+2c, \ldots$ (stopping just before b).

Zip

Another collection that can be iterated over is a *zip*. A zip is constructed from other collections all of the same length. Each element of the zip is a tuple consisting of one element from each of the input collections.

```
>>> list(zip([1,3,5],[2,4,6]))
[(1, 2), (3, 4), (5, 6)]
>>> characters = ['Neo', 'Morpheus', 'Trinity']
>>> actors = ['Keanu', 'Laurence', 'Carrie-Anne']
>>> set(zip(characters, actors))
{('Trinity', 'Carrie-Anne'), ('Neo', 'Keanu'), ('Morpheus', 'Laurence')}
>>> [character+' is played by '+actor
...     for (character,actor) in zip(characters,actors)]
['Neo is played by Keanu', 'Morpheus is played by Laurence',
 'Trinity is played by Carrie-Anne']
```

Task 0.5.19: Assign to L the list consisting of the first five letters `['A','B','C','D','E']`. Next, use L in an expression whose value is

```
[(0, 'A'), (1, 'B'), (2, 'C'), (3, 'D'), (4, 'E')]
```

Your expression should use a range and a zip, but should not use a comprehension.

Task 0.5.20: Starting from the lists `[10, 25, 40]` and `[1, 15, 20]`, write a comprehension whose value is the three-element list in which the first element is the sum of 10 and 1, the second is the sum of 25 and 15, and the third is the sum of 40 and 20. Your expression should use `zip` but not `list`.

reversed

To iterate through the elements of a list L in reverse order, use `reversed(L)`, which does not change the list L:

```
>>> [x*x for x in reversed([4, 5, 10])]
[100, 25, 16]
```

0.5.8 *Dictionaries*

We will often have occasion to use functions with finite domains. Python provides collections, called *dictionaries*, that are suitable for representing such functions. Conceptually, a dictionary is a set of key-value pairs. The syntax for specifying a dictionary in terms of its key-value pairs therefore resembles the syntax for sets—it uses curly braces—except that instead of listing the elements of the set, one lists the key-value pairs. In this syntax, each key-value pair is written using *colon* notation: an expression for the key, followed by the colon, followed by an expression for the value:

key : *value*

The function f that maps each letter in the alphabet to its rank in the alphabet could be written as

```
{'A':0, 'B':1, 'C':2, 'D':3, 'E':4, 'F':5, 'G':6, 'H':7, 'I':8,
 'J':9, 'K':10, 'L':11, 'M':12, 'N':13, 'O':14, 'P':15, 'Q':16,
 'R':17, 'S':18, 'T':19, 'U':20, 'V':21, 'W':22, 'X':23, 'Y':24,
 'Z':25}
```

As in sets, the order of the key-value pairs is irrelevant, and the keys must be immutable (no sets or lists or dictionaries). For us, the keys will mostly be integers, strings, or tuples of integers and strings.

The keys and values can be specified with expressions.

```
>>> {2+1:'thr'+'ee', 2*2:'fo'+'ur'}
{3: 'three', 4: 'four'}
```

To each key in a dictionary there corresponds only one value. If a dictionary is given multiple values for the same key, only one value will be associated with that key.

```
>>> {0:'zero', 0:'nothing'}
{0: 'nothing'}
```

Indexing into a dictionary

Obtaining the value corresponding to a particular key uses the same syntax as indexing a list or tuple: right after the dictionary expression, use square brackets around the key:

```
>>> {4:"four", 3:'three'}[4]
'four'
>>> mydict = {'Neo':'Keanu', 'Morpheus':'Laurence',
 'Trinity':'Carrie-Anne'}
>>> mydict['Neo']
'Keanu'
```

If the key is not represented in the dictionary, Python considers it an error:

```
>>> mydict['Oracle']
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
KeyError: 'Oracle'
```

Testing dictionary membership

You can check whether a key is in a dictionary using the `in` operator we earlier used for testing membership in a set:

```
>>> 'Oracle' in mydict
False
>>> mydict['Oracle'] if 'Oracle' in mydict else 'NOT PRESENT'
'NOT PRESENT'
>>> mydict['Neo'] if 'Neo' in mydict else 'NOT PRESENT'
'Keanu'
```

Lists of dictionaries

Task 0.5.21: Suppose `dlist` is a list of dictionaries and `k` is a key that appears in all the dictionaries in `dlist`. Write a comprehension that evaluates to the list whose i^{th} element is the value corresponding to key `k` in the i^{th} dictionary in `dlist`.

Test your comprehension with some data. Here are some example data.

```
dlist = [{'James':'Sean', 'director':'Terence'}, {'James':'Roger',
'director':'Lewis'}, {'James':'Pierce', 'director':'Roger'}]
k = 'James'
```

Task 0.5.22: Modify the comprehension in Task 0.5.21 to handle the case in which `k` might not appear in all the dictionaries. The comprehension evaluates to the list whose i^{th} element is the value corresponding to key `k` in the i^{th} dictionary in `dlist` if that dictionary contains that key, and `'NOT PRESENT'` otherwise.

Test your comprehension with `k = 'Bilbo'` and `k = 'Frodo'` and with the following list of dictionaries:

```
dlist = [{'Bilbo':'Ian','Frodo':'Elijah'},
         {'Bilbo':'Martin','Thorin':'Richard'}]
```

Mutating a dictionary: indexing on the left-hand side of =

You can mutate a dictionary, mapping a (new or old) key to a given value, using the syntax used for assigning a list element, namely using the index syntax on the left-hand side of an assignment:

```
>>> mydict['Agent Smith'] = 'Hugo'
>>> mydict['Neo'] = 'Philip'
>>> mydict
{'Neo': 'Philip', 'Agent Smith': 'Hugo', 'Trinity': 'Carrie-Anne',
 'Morpheus': 'Laurence'}
```

Dictionary comprehensions

You can construct a dictionary using a comprehension.

```
>>> { k:v for (k,v) in [(3,2),(4,0),(100,1)] }
{3: 2, 4: 0, 100: 1}
>>> { (x,y):x*y for x in [1,2,3] for y in [1,2,3] }
{(1, 2): 2, (3, 2): 6, (1, 3): 3, (3, 3): 9, (3, 1): 3,
 (2, 1): 2, (2, 3): 6, (2, 2): 4, (1, 1): 1}
```

Task 0.5.23: Using `range`, write a comprehension whose value is a dictionary. The keys should be the integers from 0 to 99 and the value corresponding to a key should be the square of the key.

恒等函数

The *identity function* on a set `D` is the function with the following spec:

- *input:* an element x of `D`
- *output:* x

That is, the identity function simply outputs its input.

Task 0.5.24: Assign some set to the variable `D`, e.g. `D ={'red','white','blue'}`. Now write a comprehension that evaluates to a dictionary that represents the identity function on `D`.

Task 0.5.25: Using the variables `base=10` and `digits=set(range(base))`, write a dictionary comprehension that maps each integer between zero and nine hundred ninety nine to the list of three digits that represents that integer in base 10. That is, the value should be

```
{0: [0, 0, 0], 1: [0, 0,  1], 2: [0, 0, 2], 3: [0, 0, 3], ...,
 10: [0, 1, 0], 11: [0, 1, 1], 12: [0, 1, 2], ...,
 999: [9, 9, 9]}
```

Your expression should work for any base. For example, if you instead assign 2 to base and assign `{0,1}` to `digits`, the value should be

```
{0: [0, 0, 0], 1: [0, 0, 1], 2: [0, 1, 0], 3: [0, 1, 1],
 ..., 7: [1, 1, 1]}
```

Comprehensions that iterate over dictionaries

You can write list comprehensions that iterate over the keys or the values of a dictionary, using `keys()` or `values()`:

```
>>> [2*x for x in {4:'a',3:'b'}.keys() ]
[6, 8]
>>> [x for x in {4:'a', 3:'b'}.values()]
['b', 'a']
```

Given two dictionaries `A` and `B`, you can write comprehensions that iterate over the union or intersection of the keys, using the *union* operator `|` and intersection operator `&` we learned about in Section 0.5.4.

```
>>> [k for k in {'a':1, 'b':2}.keys() | {'b':3, 'c':4}.keys()]
['a', 'c', 'b']
>>> [k for k in {'a':1, 'b':2}.keys() & {'b':3, 'c':4}.keys()]
['b']
```

Often you'll want a comprehension that iterates over the (key, value) pairs of a dictionary, using `items()`. Each pair is a tuple.

```
>>> [myitem for myitem in mydict.items()]
[('Neo', 'Philip'), ('Morpheus', 'Laurence'),
 ('Trinity', 'Carrie-Anne'), ('Agent Smith', 'Hugo')]
```

Since the items are tuples, you can access the key and value separately using unpacking:

```
>>> [k + " is played by " + v for (k,v) in mydict.items()]
['Neo is played by Philip, 'Agent Smith is played by Hugo',
'Trinity is played by Carrie-Anne', 'Morpheus is played by Laurence']
>>> [2*k+v for (k,v) in {4:0,3:2, 100:1}.items() ]
[8, 8, 201]
```

Task 0.5.26: Suppose `d` is a dictionary that maps some employee IDs (a subset of the integers from 0 to $n-1$) to salaries. Suppose `L` is an n-element list whose i^{th} element is the name of employee number i. Your goal is to write a comprehension whose value is a dictionary mapping employee names to salaries. You can assume that employee names are distinct.

Test your comprehension with the following data:

```
id2salary = {0:1000.0, 3:990, 1:1200.50}
names = ['Larry', 'Curly', '', 'Moe']
```

0.5.9 *Defining one-line procedures*

The procedure *twice* : $\mathbb{R} \longrightarrow \mathbb{R}$ that returns twice its input can be written in Python as follows:

```
def twice(z): return 2*z
```

The word `def` introduces a procedure definition. The name of the function being defined is `twice`. The variable `z` is called the *formal argument* to the procedure. Once this procedure is defined, you can invoke it using the usual notation: the name of the procedure followed by an expression in parenthesis, e.g. `twice(1+2)`

The value 3 of the expression `1+2` is the *actual argument* to the procedure. When the procedure is invoked, the formal argument (the variable) is temporarily bound to the actual argument, and the body of the procedure is executed. At the end, the binding of the actual argument is removed. (The binding was temporary.)

Task 0.5.27: Try entering the definition of `twice(z)`. After you enter the definition, you will see the ellipsis. Just press enter. Next, try invoking the procedure on some actual arguments. Just for fun, try strings or lists. Finally, verify that the variable `z` is now not bound to any value by asking Python to evaluate the expression consisting of `z`.

Task 0.5.28: Define a one-line procedure `nextInts(L)` specified as follows:

- *input:* list L of integers
- *output:* list of integers whose i^{th} element is one more than the i^{th} element of L
- *example:* input $[1, 5, 7]$, output $[2, 6, 8]$.

Task 0.5.29: Define a one-line procedure `cubes(L)` specified as follows:

- *input:* list L of numbers
- *output:* list of numbers whose i^{th} element is the cube of the i^{th} element of L
- *example:* input $[1, 2, 3]$, output $[1, 8, 27]$.

Task 0.5.30: Define a one-line procedure `dict2list(dct,keylist)` with this spec:

- *input:* dictionary *dct*, list *keylist* consisting of the keys of *dct*
- *output:* list L such that $L[i]$ = *dct*[keylist[i]] for $i = 0, 1, 2, \ldots,$ len(*keylist*) $- 1$
- *example:* input *dct*=`{'a':'A', 'b':'B', 'c':'C'}` and *keylist*=`['b','c','a']`, output `['B', 'C', 'A']`

Task 0.5.31: Define a one-line procedure `list2dict(L, keylist)` specified as follows:

- *input:* list L, list *keylist* of immutable items
- *output:* dictionary that maps keylist[i] to $L[i]$ for $i = 0, 1, 2, \ldots,$ len(L) $- 1$
- *example:* input L=`['A','B','C']` and *keylist*=`['a','b','c']`, output `{'a':'A', 'b':'B', 'c':'C'}`

Hint: Use a comprehension that iterates over a zip or a range.

Task 0.5.32: Write a procedure `all_3_digit_numbers(base, digits)` with the following spec:

- *input:* a positive integer *base* and the set *digits* which should be $\{0, 1, 2, \ldots, base-1\}$.
- *output:* the set of all three-digit numbers where the base is *base*

For example,

```
>>> all_3_digit_numbers(2, {0,1})
{0, 1, 2, 3, 4, 5, 6, 7}
>>> all_3_digit_numbers(3, {0,1,2})
{0, 1, 2, 3, 4, 5, 6, 7, 8, 9, 10, 11, 12, 13, 14, 15, 16, 17, 18,
 19, 20, 21, 22, 23, 24, 25, 26}
>>> all_3_digit_numbers(10, {0,1,2,3,4,5,6,7,8,9})
{0, 1, 2, 3, 4, 5, 6, 7, 8, 9, 10, 11, 12, 13, 14, 15, 16, 17, 18,
 19, 20, 21, 22, 23, 24, 25, 26, 27, 28, 29, 30, 31, 32, 33, 34, 35,
   ...
985, 986, 987, 988, 989, 990, 991, 992, 993, 994, 995, 996, 997, 998, 999}
```

0.6 *Lab. Python—modules and control structures—and inverse index*

In this lab, you will create a simple search engine. One procedure will be responsible for reading in a large collection of documents and indexing them to facilitate quick responses to subsequent search queries. Other procedures will use the index to answer the search queries.

The main purpose of this lab is to give you more Python programming practice.

0.6.1 *Using existing modules*

Python comes with an extensive library, consisting of components called *modules*. In order to use the definitions defined in a module, you must either import the module itself or import the specific definitions you want to use from the module. If you import the module, you must refer to a procedure or variable defined therein by using its *qualified name*, i.e. the name of the module followed by a dot followed by the short name.

For example, the library `math` includes many mathematical procedures such as square-root, cosine, and natural logarithm, and mathematical constants such as π and e.