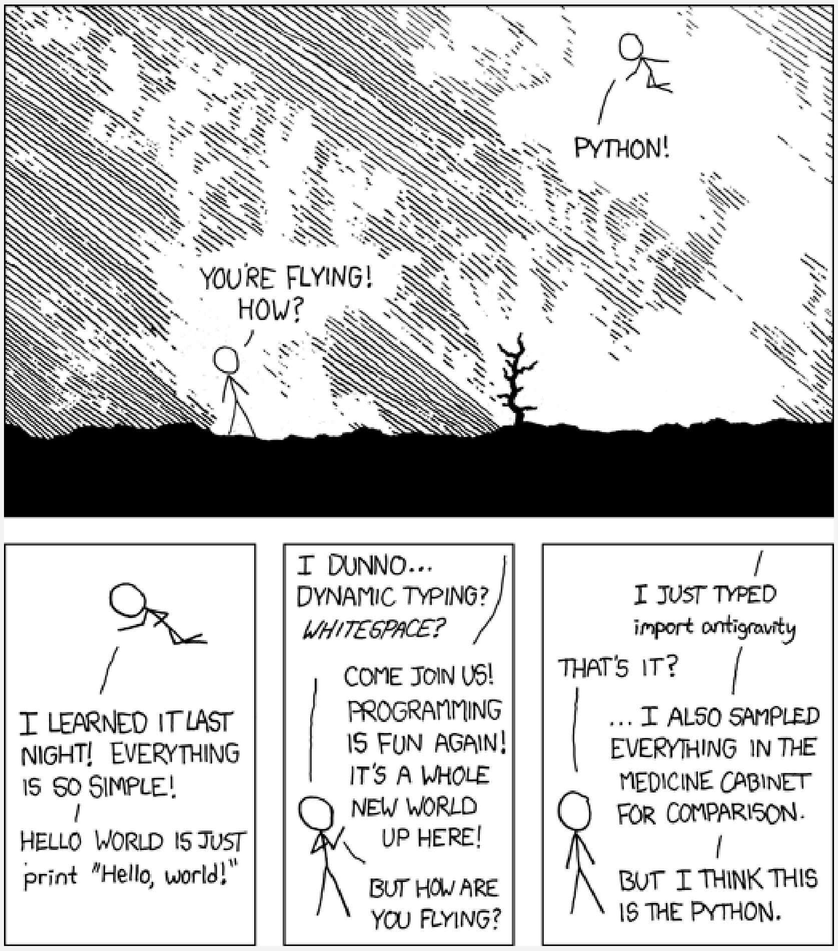
0.5 *Lab: Introduction to Python—sets, lists, dictionaries, and comprehensions*


PYTHON!
YOU'RE FLYING! HOW?
I LEARNED IT LAST NIGHT! EVERYTHING IS SO SIMPLE!
HELLO WORLD IS JUST print "Hello, world!"
I DUNNO... DYNAMIC TYPING? WHITESPACE?
COME JOIN US! PROGRAMMING IS FUN AGAIN! IT'S A WHOLE NEW WORLD UP HERE!
BUT HOW ARE YOU FLYING?
I JUST TYPED import antigravity
THAT'S IT?
... I ALSO SAMPLED EVERYTHING IN THE MEDICINE CABINET FOR COMPARISON.
BUT I THINK THIS IS THE PYTHON.

Python `http://xkcd.com/353/`

We will be writing all our code in Python (Version 3.x). In writing Python code, we emphasize the use of *comprehensions*, which allow one to express computations over the elements of a set, list, or dictionary without a traditional for-loop. Use of comprehensions leads to more compact and more readable code, code that more clearly expresses the mathematical idea behind the computation being expressed. Comprehensions might be new to even some readers who are familiar with Python, and we encourage those readers to at least skim the material on this topic.

To start Python, simply open a console (also called a shell or a terminal or, under Windows, a "Command Prompt" or "MS-DOS Prompt"), and type `python3` (or perhaps just `python`) to the console (or shell or terminal or Command Prompt) and hit the `Enter` key. After a few lines telling you what version you are using (e.g., Python 3.3.3), you should see `>>>` followed by a space. This is the *prompt*; it indicates that Python is waiting for you to type something. When you type an expression and hit the `Enter` key, Python evaluates the expression and prints the result, and then prints another prompt. To get out of this environment, type `quit()` and `Enter`, or `Control-D`. To interrupt Python when it is running too long, type `Control-C`.

This environment is sometimes called a *REPL*, an acronym for "read-eval-print loop." It reads what you type, evaluates it, and prints the result if any. In this assignment, you will interact with Python primarily through the REPL. In each task, you are asked to come up with an expression of a certain form.

There are two other ways to run Python code. You can import a *module* from within the REPL, and you can run a Python script from the command line (outside the REPL). We will discuss modules and importing in the next lab assignment. This will be an important part of your interaction with Python.

0.5.1 *Simple expressions*

Arithmetic and numbers

You can use Python as a calculator for carrying out arithmetic computations. The binary operators +, *, -, / work as you would expect. To take the negative of a number, use - as a unary operator (as in -9). Exponentiation is represented by the binary operator **, and truncating integer division is //. Finding the remainder when one integer is divided by another (modulo) is done using the % operator. As usual, ** has precedence over * and / and //, which have precedence over + and -, and parentheses can be used for grouping.

To get Python to carry out a calculation, type the expression and press the Enter/Return key:

```
>>> 44+11*4-6/11.
87.454545454545454
>>>
```

Python prints the answer and then prints the prompt again.

Task 0.5.1: Use Python to find the number of minutes in a week.

Task 0.5.2: Use Python to find the remainder of 2304811 divided by 47 without using the modulo operator %. (Hint: Use //.)

Python uses a traditional programming notation for scientific notation. The notation `6.022e23` denotes the value 6.02×10^{23}, and `6.626e-34` denotes the value 6.626×10^{-34}. As we will discover, since Python uses limited-precision arithmetic, there are round-off errors:

```
>>> 1e16 + 1
1e16
```

Strings

A string is a series of characters that starts and ends with a single-quote mark. Enter a string, and Python will repeat it back to you:

```
>>> 'This sentence is false.'
'This sentence is false.'
```

You can also use double-quote marks; this is useful if your string itself contains a single-quote mark:

```
>>> "So's this one."
"So's this one."
```

Python is doing what it usually does: it *evaluates* (finds the value of) the expression it is given and prints the value. The value of a string is just the string itself.

Comparisons and conditions and Booleans

You can compare values (strings and numbers, for example) using the operators ==, < , >, <=, >=, and !=. (The operator != is inequality.)

```
>>> 5 == 4
False
>>> 4 == 4
True
```

The value of such a comparison is a Boolean value (True or False). An expression whose value is a boolean is called a Boolean expression.

Boolean operators such as `and` and `or` and `not` can be used to form more complicated Boolean expressions.

```
>> True and False
False
>>> True and not (5 == 4)
True
```

Task 0.5.3: Enter a Boolean expression to test whether the sum of 673 and 909 is divisible by 3.

0.5.2 *Assignment statements*

The following is a *statement*, not an expression. Python executes it but produces neither an error message nor a value.

```
>>> mynum = 4+1
```

The result is that henceforth the variable `mynum` is bound to the value 5. Consequently, when Python evaluates the expression consisting solely of `mynum`, the resulting value is 5. We say therefore that the value of `mynum` is 5.

A bit of terminology: the variable being assigned to is called the *left-hand side* of an assignment, and the expression whose value is assigned is called the *right-hand side*.

A variable name must start with a letter and must exclude certain special symbols such as the dot (period). The underscore _ is allowed in a variable name. A variable can be bound to a value of any type. You can rebind `mynum` to a string:

```
>>> mynum = 'Brown'
```

This binding lasts until you assign some other value to `mynum` or until you end your Python session. It is called a *top-level* binding. We will encounter cases of binding variables to values where the bindings are temporary.

It is important to remember (and second nature to most experienced programmers) that an assignment statement binds a variable to the *value* of an expression, not to the expression itself. Python first evaluates the right-hand side and only then assigns the resulting value to the left-hand side. This is the behavior of most programming languages.

Consider the following assignments.

```
>>> x = 5+4
>>> y = 2 * x
>>> y
18
>>> x = 12
>>> y
18
```

In the second assignment, `y` is assigned the value of the expression `2 * x`. The value of that expression is 9, so `y` is bound to 18. In the third assignment, `x` is bound to 12. This does not change the fact that `y` is bound to 18.

0.5.3 *Conditional expressions*

There is a syntax for conditional expressions:

$\langle$*expression*$\rangle$ `if` $\langle$*condition*$\rangle$ `else` $\langle$*expression*$\rangle$

The *condition* should be a Boolean expression. Python evaluates the condition; depending on whether it is `True` or `False`, Python then evaluates either the first or second *expression*, and uses the result as the result of the entire conditional expression.

For example, the value of the expression `x if x>0 else -x` is the absolute value of `x`.

Task 0.5.4: Assign the value -9 to `x` and 1/2 to `y`. Predict the value of the following expression, then enter it to check your prediction:

```
2**(y+1/2) if x+10<0 else 2**(y-1/2)
```

0.5.4 *Sets*

Python provides some simple data structures for grouping together multiple values, and integrates them with the rest of the language. These data structures are called *collections*. We start with sets.

A set is an unordered collection in which each value occurs at most once. You can use curly braces to give an expression whose value is a set. Python prints sets using curly braces.

```
>>> {1+2, 3, "a"}
{'a', 3}
>>> {2, 1, 3}
{1, 2, 3}
```

Note that duplicates are eliminated and that the order in which the elements of the output are printed does not necessarily match the order of the input elements.

The *cardinality* of a set S is the number of elements in the set. In Mathese we write $|S|$ for the cardinality of set S. In Python, the cardinality of a set is obtained using the procedure `len(·)`.

```
>>> len({'a', 'b', 'c', 'a', 'a'})
3
```

Summing

The sum of elements of collection of values is obtained using the procedure `sum(·)`.

```
>>> sum({1,2,3})
6
```

If for some reason (we'll see one later) you want to start the sum not at zero but at some other value, supply that value as a second argument to `sum(·)`:

```
>>> sum({1,2,3}, 10)
16
```

Testing set membership

Membership in a set can be tested using the `in` operator and the `not in` operator. If S is a set, x `in` S is a Boolean expression that evaluates to `True` if the value of x is a member of the set S, and `False` otherwise. The value of a `not in` expression is just the opposite

```
>>> S={1,2,3}
>>> 2 in S
True
>>> 4 in S
False
>>> 4 not in S
True
```

Set union and intersection

The *union* of two sets S and T is a new set that contains every value that is a member of S or a member of T (or both). Python uses the vertical bar `|` as the *union* operator:

```
>>> {1,2,3} | {2,3,4}
{1, 2, 3, 4}
```

The *intersection* of S and T is a new set that contains every value that is a member of both S and T. Python uses the ampersand `&` as the *intersection* operator:

```
>>> {1,2,3} & {2,3,4}
{2, 3}
```

Mutating a set

A value that can be altered is a *mutable* value. Sets are mutable; elements can be added and removed using the `add` and `remove` methods:

```
>>> S={1,2,3}
>>> S.add(4)
>>> S.remove(2)
>>> S
{1, 3, 4}
```

The syntax using the dot should be familiar to students of object-oriented programming languages such as Java and C++. The operations `add(·)` and `remove(·)` are *methods*. You can think of a method as a procedure that takes an extra argument, the value of the expression to the left of the dot.

Python provides a method `update(...)` to add to a set all the elements of another collection (e.g. a set or a list):

```
>>> S.update({4, 5, 6})
>>> S
{1, 3, 4, 5, 6}
```

Similarly, one can intersect a set with another collection, removing from the set all elements not in the other collection:

```
>>> S.intersection_update({5,6,7,8,9})
>>> S
{5, 6}
```

Suppose two variables are bound to the same value. A mutation to the value made through one variable is seen by the other variable.

```
>>> T=S
>>> T.remove(5)
>>> S
{6}
```

This behavior reflects the fact that Python stores only one copy of the underlying data structure. After Python executes the assignment statement `T=S`, both `T` and `S` point to the same data structure. This aspect of Python will be important to us: many different variables can point to the same huge set without causing a blow-up of storage requirements.

Python provides a method for copying a collection such as a set:

```
>>> U=S.copy()
>>> U.add(5)
>>> S
{1, 3}
```

The assignment statement binds `U` not to the value of `S` but to a copy of that value, so mutations to the value of `U` don't affect the value of `S`.

Set comprehensions

Python provides for expressions called *comprehensions* that let you build collections out of other collections. We will be using comprehensions a lot because they are useful in constructing an expression whose value is a collection, and they mimic traditional mathematical notation. Here's an example:

```
>>> {2*x for x in {1,2,3} }
{2, 4, 6}
```

This is said to be a set comprehension *over* the set `{1,2,3}`. It is called a set comprehension because its value is a set. The notation is similar to the traditional mathematical notation for expressing sets in terms of other sets, in this case $\{2x \ : \ x \in \{1,2,3\}\}$. To compute the value, Python iterates over the elements of the set `{1,2,3}`, temporarily binding the control variable `x` to each element in turn and evaluating the expression `2*x` in the context of that binding. Each of the values obtained is an element of the final set. (The bindings of `x` during the evaluation of the comprehension do not persist after the evaluation completes.)

Task 0.5.5: Write a comprehension over $\{1,2,3,4,5\}$ whose value is the set consisting of the squares of the first five positive integers.

Task 0.5.6: Write a comprehension over $\{0,1,2,3,4\}$ whose value is the set consisting of the first five powers of two, starting with 2^0.

Using the union operator `|` or the intersection operator `&`, you can write set expressions for the union or intersection of two sets, and use such expressions in a comprehension:

```
>>> {x*x for x in S | {5, 7}}
{1, 25, 49, 9}
```

By adding the phrase `if` ⟨*condition*⟩ at the end of the comprehension (before the closing brace "}"), you can skip some of the values in the set being iterated over:

```
>>> {x*x for x in S | {5, 7}  if x > 2}
{9, 49, 25}
```

I call the conditional clause a *filter*.

You can write a comprehension that iterates over the Cartesian product of two sets:

```
>>>{x*y for x in {1,2,3} for y in {2,3,4}}
{2, 3, 4, 6, 8, 9, 12}
```

This comprehension constructs the set of the products of every combination of `x` and `y`. I call this a *double comprehension.*

Task 0.5.7: The value of the previous comprehension,

```
{x*y for x in {1,2,3} for y in {2,3,4}}
```

is a seven-element set. Replace `{1,2,3}` and `{2,3,4}` with two other three-element sets so that the value becomes a nine-element set.

Here is an example of a double comprehension with a filter:

```
>>> {x*y for x in {1,2,3} for y in {2,3,4} if x != y}
{2, 3, 4, 6, 8, 12}
```

Task 0.5.8: Replace `{1,2,3}` and `{2,3,4}` in the previous comprehension with two disjoint (i.e. non-overlapping) three-element sets so that the value becomes a five-element set.

Task 0.5.9: Assume that `S` and `T` are assigned sets. Without using the *intersection operator* `&`, write a comprehension over `S` whose value is the intersection of `S` and `T`. Hint: Use a membership test in a filter at the end of the comprehension.

Try out your comprehension with `S = {1,2,3,4}` and `T = {3,4,5,6}`.

Remarks

The empty set is represented by `set()`. You would think that `{}` would work but, as we will see, that notation is used for something else.

You cannot make a set that has a set as element. This has nothing to do with Cantor's Paradox—-Python imposes the restriction that the elements of a set must not be mutable, and sets are mutable. The reason for this restriction will be clear to a student of data structures from the error message in the following example:

```
>>> {{1,2},3}
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
TypeError: unhashable type: 'set'
```

There is a nonmutable version of set called *frozenset.* Frozensets can be elements of sets. However, we won't be using them.

0.5.5 *Lists*

Python represents sequences of values using *lists.* In a list, order is significant and repeated elements are allowed. The notation for lists uses square brackets instead of curly braces. The empy list is represented by `[]`.

```
>>> [1,1+1,3,2,3]
[1, 2, 3, 2, 3]
```

There are no restrictions on the elements of lists. A list can contain a set or another list.

```
>>> [[1,1+1,4-1],{2*2,5,6}, "yo"]
[[1, 2, 3], {4, 5, 6}, 'yo']
```

However, a set cannot contain a list since lists are mutable.

The *length* of a list, obtained using the procedure `len(·)`, is the number of elements in the list, even though some of those elements may themselves be lists, and even though some elements might have the same value:

```
>>> len([[1,1+1,4-1],{2*2,5,6}, "yo", "yo"])
4
```

As we saw in the section on sets, the sum of elements of a collection can be computed using `sum(·)`

```
>>> sum([1,1,0,1,0,1,0])
4
>>> sum([1,1,0,1,0,1,0], -9)
-5
```

In the second example, the second argument to `sum(·)` is the value to start with.

Task 0.5.10: Write an expression whose value is the average of the elements of the list `[20, 10, 15, 75]`.

List concatenation

You can combine the elements in one list with the elements in another list to form a new list (without changing the original lists) using the + operator.

```
>>> [1,2,3]+["my", "word"]
[1, 2, 3, 'my', 'word']
>>> mylist = [4,8,12]
>>> mylist + ["my", "word"]
[4, 8, 12, 'my', 'word']
>>> mylist
[4, 8, 12]
```

You can use `sum(·)` on a collection of lists, obtaining the concatenation of all the lists, by providing `[]` as the second argument.

```
>>> sum([ [1,2,3], [4,5,6], [7,8,9] ])
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
TypeError: unsupported operand type(s) for +: 'int' and 'list'
>>> sum([ [1,2,3], [4,5,6], [7,8,9] ], [])
[1, 2, 3, 4, 5, 6, 7, 8, 9]
```

List comprehensions

Next we discuss how to write a list comprehension (a comprehension whose value is a list). In the following example, a list is constructed by iterating over the elements in a set.

```
>>> [2*x for x in {2,1,3,4,5} ]
[2, 4, 6, 8, 10]
```

Note that the order of elements in the resulting list might not correspond to the order of elements in the set since the latter order is not significant.

You can also use a comprehension that constructs a list by iterating over the elements in a list:

```
>>> [ 2*x for x in [2,1,3,4,5] ]
[4, 2, 6, 8, 10]
```

Note that the list [2,1,3,4,5] specifies the order among its elements. In evaluating the comprehension Python iterates through them in that order. Therefore the order of elements in the resulting list corresponds to the order in the list iterated over.

You can also write list comprehensions that iterate over multiple collections using two control variables. As I mentioned in the context of sets, I call these "double comprehensions". Here is an example of a list comprehension over two lists.

```
>>> [ x*y for x in [1,2,3] for y in [10,20,30] ]
[10, 20, 30, 20, 40, 60, 30, 60, 90]
```

The resulting list has an element for every combination of an element of [1,2,3] with an element of [10,20,30].

We can use a comprehension over two sets to form the Cartesian product.

Task 0.5.11: Write a double list comprehension over the lists ['A','B','C'] and [1,2,3] whose value is the list of all possible two-element lists [letter, number]. That is, the value is

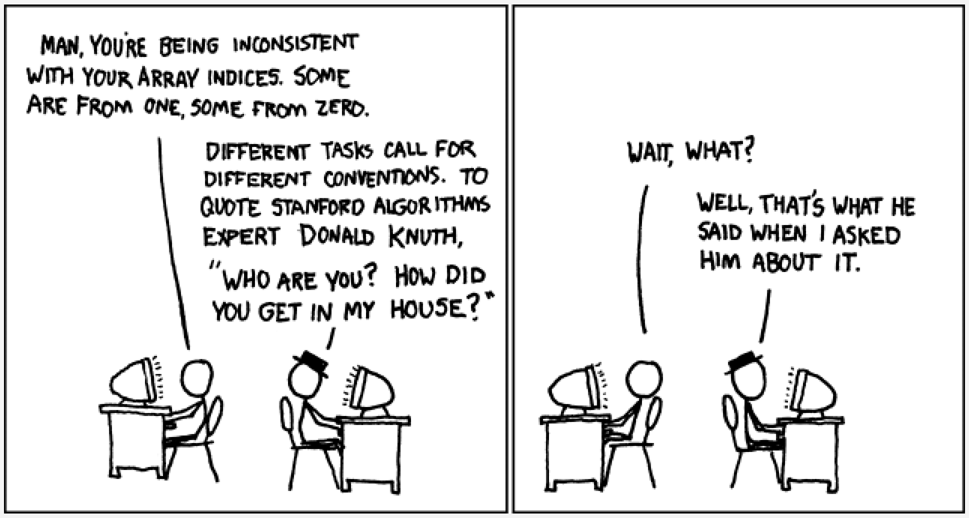
```
[['A', 1], ['A', 2], ['A', 3], ['B', 1], ['B', 2],['B', 3],
['C', 1], ['C', 2], ['C', 3]]
```

Task 0.5.12: Suppose LofL has been assigned a list whose elements are themselves lists of numbers. Write an expression that evaluates to the sum of all the numbers in all the lists. The expression has the form

```
sum([sum(...
```

and includes one comprehension. Test your expression after assigning [[.25, .75, .1], [-1, 0], [4, 4, 4, 4]] to LofL. Note that your expression should work for a list of any length.

Obtaining elements of a list by indexing


MAN, YOU'RE BEING INCONSISTENT WITH YOUR ARRAY INDICES. SOME ARE FROM ONE, SOME FROM ZERO.
DIFFERENT TASKS CALL FOR DIFFERENT CONVENTIONS. TO QUOTE STANFORD ALGORITHMS EXPERT DONALD KNUTH, "WHO ARE YOU? HOW DID YOU GET IN MY HOUSE?"
WAIT, WHAT?
WELL, THAT'S WHAT HE SAID WHEN I ASKED HIM ABOUT IT.

Donald Knuth http://xkcd.com/163/

There are two ways to obtain an individual element of a list. The first is by indexing. As in some other languages (Java and c++, for example) indexing is done using square brackets around the index. Here is an example. Note that the first element of the list has index 0.

```
>>> mylist[0]
4
```

```
>>> ['in','the','CIT'][1]
'the'
```

Slices: A *slice* of a list is a new list consisting of a consecutive subsequence of elements of the old list, namely those indexed by a range of integers. The range is specified by a colon-separated pair $i : j$ consisting of the index i as the first element and j as one past the index of the last element. Thus `mylist[1:3]` is the list consisting of elements 1 and 2 of `mylist`.

Prefixes: If the first element i of the pair is 0, it can be omitted, so `mylist[:2]` consists of the first 2 elements of `mylist`. This notation is useful for obtaining a prefix of a list.

Suffixes: If the second element j of the pair is the length of the list, it can be omitted, so `mylist[1:]` consists of all elements of `mylist` except element 0.

```
>>> L = [0,10,20,30,40,50,60,70,80,90]
>>> L[:5]
[0, 10, 20, 30, 40]
>>> L[5:]
[50, 60, 70, 80, 90]
```

Slices that skip You can use a colon-separated *triple* $a:b:c$ if you want the slice to include every c^{th} element. For example, here is how you can extract from `L` the list consisting of even-indexed elements and the list consisting of odd-indexed elements:

```
>>> L[::2]
[0, 20, 40, 60, 80]
>>> L[1::2]
[10, 30, 50, 70, 90]
```

Obtaining elements of a list by unpacking

The second way to obtain individual elements is by *unpacking*. Instead of assigning a list to a single variable as in `mylist =[4,8,12]`, one can assign to a list of variables:

```
>>> [x,y,z] = [4*1, 4*2, 4*3]
>>> x
4
>>> y
8
```

I called the left-hand side of the assignment a “list of variables,” but beware: this is a notational fiction. Python does not allow you to create a value that is a list of variables. The assignment is simply a convenient way to assign to each of the variables appearing in the left-hand side.

Task 0.5.13: Find out what happens if the length of the left-hand side list does not match the length of the right-hand side list.

Unpacking can similarly be used in comprehensions:

```
>>> listoflists = [[1,1],[2,4],[3, 9]]
>>> [y for [x,y] in listoflists]
[1, 4, 9]
```

Here the two-element list `[x,y]` iterates over all elements of `listoflists`. This would result in an error message if some element of `listoflists` were not a two-element list.

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

You can mutate a list, replacing its i^{th} element, using indexing on the left-hand side of the =, analogous to an assignment statement:

```
>>> mylist = [30, 20, 10]
>>> mylist[1] = 0
>>> mylist
[30, 0, 10]
```

Slices can also be used on the left-hand side but we will not use this.

0.5.6 *Tuples*

Like a list, a tuple is an ordered sequence of elements. However, tuples are immutable so they can be elements of sets. The notation for tuples is the same as that for lists except that ordinary parentheses are used instead of square brackets.

```
>>> (1,1+1,3)
(1, 2, 3)
>>> {0, (1,2)} | {(3,4,5)}
{(1, 2), 0, (3, 4, 5)}
```

Obtaining elements of a tuple by indexing and unpacking

You can use indexing to obtain an element of a tuple.

```
>>> mytuple = ("all", "my", "books")
>>> mytuple[1]
'my'
>>> (1, {"A", "B"}, 3.14)[2]
3.14
```

You can also use unpacking with tuples. Here is an example of top-level variable assignment:

```
>>> (a,b) = (1,5-3)
>>> a
1
```

In some contexts, you can get away without the parentheses, e.g.

```
>>> a,b = (1,5-3)
```

or even

```
>>> a,b = 1,5-3
```

You can use unpacking in a comprehension:

```
>>> [y for (x,y) in [(1,'A'),(2,'B'),(3,'C')] ]
['A', 'B', 'C']
```

Task 0.5.14: Suppose S is a set of integers, e.g. $\{-4, -2, 1, 2, 5, 0\}$. Write a triple comprehension whose value is a list of all three-element tuples (i, j, k) such that i, j, k are elements of S whose sum is zero.

Task 0.5.15: Modify the comprehension of the previous task so that the resulting list does not include $(0, 0, 0)$. Hint: add a filter.

Task 0.5.16: Further modify the expression so that its value is not the list of all such tuples but is the first such tuple.

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(range(10))
{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 range(10))` 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.

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. However, not every employee ID is represented in d.

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 $L[i] = dct[\text{keylist}[i]]$ for $i = 0, 1, 2, \ldots, \text{len}(keylist) - 1$
- *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 $i = 0, 1, 2, \ldots, \text{len}(L) - 1$
- *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.

Task 0.6.1: Import the `math` module using the command

```
>>> import math
```

Call the built-in procedure `help(modulename)` on the module you have just imported:

```
>>> help(math)
```

This will cause the console to show documentation on the module. You can move forward by typing `f` and backward by typing `b`, and you can quit looking at the documentation by typing `q`.

Use procedures defined by the `math` module to compute the square root of 3, and raise it to the power of 2. The result might not be what you expect. Keep in mind that Python represents nonintegral real numbers with limited precision, so the answers it gives are only approximate.

Next compute the square root of -1, the cosine of π, and the natural logarithm of e.

The short name of the square-root function is `sqrt` so its qualified name is `math.sqrt`. The short names of the cosine and the natural logarithm are `cos` and `log`, and the short names of π and e are `pi` and `e`.

The second way to bring a procedure or variable from a module into your Python environment is to specifically import the item itself from the module, using the syntax

`from` ⟨*module name*⟩ `import` ⟨*short name*⟩

after which you can refer to it using its short name.

Task 0.6.2: The module `random` defines a procedure `randint(a,b)` that returns an integer chosen uniformly at random from among $\{a, a+1, \ldots, b\}$. Import this procedure using the command

```
>>> from random import randint
```

Try calling `randint` a few times. Then write a one-line procedure `movie_review(name)` that takes as argument a string naming a movie, and returns a string review selected uniformly at random from among two or more alternatives (Suggestions: "See it!", "A gem!", "Ideological claptrap!")

0.6.2 *Creating your own modules*

You can create your own modules simply by entering the text of your procedure definitions and variable assignments in a file whose name consists of the module name you choose, followed by `.py`. Use a text editor such as kate or vim or, my personal favorite, emacs.

The file can itself contain import statements, enabling the code in the file to make use of definitions from other modules.

If the file is in the current working directory when you start up Python, you can import the module.[a]

Task 0.6.3: In Tasks 0.5.30 and 0.5.31 of Lab 0.5, you wrote procedures `dict2list(dct, keylist)` and `list2dict(L, keylist)`. Download the file `dictutil.py` from `http://resources.codingthematrix.com`. (That site hosts support code and sample data for the problems in this book.) Edit the provided file `dictutil.py` and edit it, replacing each occurence of pass with the appropriate statement. Import this module, and test the procedures. We will have occasion to use this module in the future.

[a]There is an environment variable, `PYTHONPATH`, that governs the sequence of directories in which Python searches for modules.

Reloading

You will probably find it useful when debugging your own module to be able to edit it and load the edited version into your current Python session. Python provides the procedure `reload(module)` in the module `imp`. To import this procedure, use the command

```
>>> from imp import reload
```

Note that if you import a specific definition using the `from ... import ...` syntax then you cannot reload it.

Task 0.6.4: Edit `dictutil.py`. Define a procedure `listrange2dict(L)` with this spec:

- *input:* a list L
- *output:* a dictionary that, for $i = 0, 1, 2, \ldots, \text{len}(L) - 1$, maps i to $L[i]$

You can write this procedure from scratch or write it in terms of `list2dict(L, keylist)`.
Use the statement

```
>>> reload(dictutil)
```

to reload your module, and then test `listrange2dict` on the list `['A','B','C']`.

0.6.3 *Loops and conditional statements*

Comprehensions are not the only way to loop over elements of a set, list, dictionary, tuple, range, or zip. For the traditionalist programmer, there are *for-loops*: `for x in {1,2,3}: print(x)`. In this statement, the variable `x` is bound to each of the elements of the set in turn, and the statement `print(x)` is executed in the context of that binding.

There are also *while-loops*: `while v[i] == 0: i = i+1`.

There are also conditional statements (as opposed to conditional expressions):
`if x > 0: print("positive")`

0.6.4 *Grouping in Python using indentation*

You will sometimes need to define loops or conditional statements in which the body consists of more than one statement. Most programming languages have a way of grouping a series of statements into a block. For example, c and Java use curly braces around the sequence of statements.

Python uses *indentation* to indicate grouping of statements. **All the statements forming a block should be indented the same number of spaces.** Python is very picky about this. Python files we provide will use **four** spaces to indent. Also, don't mix tabs with spaces in the same block. In fact, I recommend you avoid using tabs for indentation with Python.

Statements at the top level should have no indentation. The group of statements forming the body of a control statement should be indented more than the control statement. Here's an example:

```
for x in [1,2,3]:
  y = x*x
  print(y)
```

This prints 1, 4, and 9. (After the loop is executed, `y` remains bound to 9 and `x` remains bound to 3.)

Task 0.6.5: Type the above for-loop into Python. You will see that, after you enter the first line, Python prints an ellipsis (...) to indicate that it is expecting an indented block of statements. Type a space or two before entering the next line. Python will again print the ellipsis. Type a space or two (same number of spaces as before) and enter the next line. Once again Python will print an ellipsis. Press *enter*, and Python should execute the loop.

The same use of indentation can be used used in conditional statements and in procedure definitions.

```
def quadratic(a,b,c):
   discriminant = math.sqrt(b*b - 4*a*c)
   return ((-b + discriminant)/(2*a), (-b - discriminant)/(2*a))
```

You can nest as deeply as you like:

```
def print_greater_quadratic(L):
  for a, b, c in L:
    plus, minus = quadratic(a, b, c)
    if plus > minus:
      print(plus)
    else:
      print(minus)
```

Many text editors help you handle indentation when you write Python code. For example, if you are using emacs to edit a file with a `.py` suffix, after you type a line ending with a colon and hit return, emacs will automatically indent the next line the proper amount, making it easy for you to start entering lines belonging to a block. After you enter each line and hit Return, emacs will again indent the next line. However, emacs doesn't know when you have written the last line of a block; when you need to write the first line outside of that block, you should hit Delete to unindent.

0.6.5 *Breaking out of a loop*

As in many other programming languages, when Python executes the `break` statement, the loop execution is terminated, and execution continues immediately after the innermost nested loop containing the statement.

```
>>> s = "There is no spoon."
>>> for i in range(len(s)):
...   if s[i] == 'n':
...     break
...
>>> i
9
```

0.6.6 *Reading from a file*

In Python, a *file* object is used to refer to and access a file. The expression `open('stories_small.txt')` returns a file object that allows access to the file with the name given. You can use a comprehension or for-loop to loop over the lines in the file

```
>>> f = open('stories_big.txt')
>>> for line in f:
...  print(line)
```

or, if the file is not too big, use `list(·)` to directly obtain a list of the lines in the file, e.g.

```
>>> f = open('stories_small.txt')
>>> stories = list(f)
>>> len(stories)
50
```

In order to read from the file again, one way is to first create a new file object by calling `open` again.

0.6.7 *Mini-search engine*

Now, for the core of the lab, you will be writing a program that acts as a sort of search engine.

Given a file of "documents" where each document occupies a line of the file, you are to build a data structure (called an *inverse index*) that allows you to identify those documents containing a given word. We will identify the documents by *document number*: the document represented by the first line of the file is document number 0, that represented by the second line is document number 1, and so on.

You can use a method defined for strings, `split()`, which splits the string at spaces into substrings, and returns a list of these substrings:

```
>>> mystr = 'Ask not what you can do for your country.'
>>> mystr.split()
['Ask', 'not', 'what', 'you', 'can', 'do', 'for', 'your', 'country.']
```

Note that the period is considered part of a substring. To make this lab easier, we have prepared a file of documents in which punctuation are separated from words by spaces.

Often one wants to iterate through the elements of a list while keeping track of the indices of the elements. Python provides `enumerate(L)` for this purpose.

```
>>> list(enumerate(['A','B','C']))
[(0, 'A'), (1, 'B'), (2, 'C')]
>>> [i*x for (i,x) in enumerate([10,20,30,40,50])]
[0, 20, 60, 120, 200]
>>> [i*s for (i,s) in enumerate(['A','B','C','D','E'])]
['', 'B', 'CC', 'DDD', 'EEEE']
```

Task 0.6.6: Write a procedure `makeInverseIndex(strlist)` that, given a list of strings (documents), returns a dictionary that maps each word to the set consisting of the document numbers of documents in which that word appears. This dictionary is called an *inverse index*. (Hint: use `enumerate`.)

Task 0.6.7: Write a procedure `orSearch(inverseIndex, query)` which takes an inverse index and a list of words `query`, and returns the set of document numbers specifying all documents that conain *any* of the words in `query`.

Task 0.6.8: Write a procedure `andSearch(inverseIndex, query)` which takes an inverse index and a list of words `query`, and returns the set of document numbers specifying all documents that contain *all* of the words in `query`.

Try out your procedures on these two provided files:

- `stories_small.txt`
- `stories_big.txt`

0.7 Review questions

- What does the notation $f : A \longrightarrow B$ mean?
- What are the criteria for f to be an invertible function?
- What is associativity of functional composition?
- What are the criteria for a function to be a probability function?
- What is the Fundamental Principle of Probability Theory?
- If the input to an invertible function is chosen randomly according to the uniform distribution, what is the distribution of the output?