# Lab 05: Iterables and Higher-Order Functions

## Contents



## Objectives

By the end of this lab you will understand:

- The differences between different iterable types

By the end of this lab you will be able to:

- Iterate in several different (and efficient) ways
- Apply functions to multiple objects at once
- Filter out objects of interest

## Instructions

This lab handout is a bit different, as this is the last lab covering features of Python as a language; in the second half of the module we'll switch to looking at all the cool things we can do with Python, rather than how to write it. This means it contains a lot of extra information on different types and techniques you may find useful later on, so will hopefully serve as a good reference. It also means that you may find it best to skip to Section 2 to start, and then go back and look at Section 1 only if/when you find it useful.

## 1 Iterables

Each of the sequence types — lists, dictionaries, and strings — that we've seen thus far has their own set of circumstances under which they're best applied, but the one thing they all have in common is *iteration*. You've already seen this, since it is possible to loop through these objects using `for` loops. There are two other types we'll introduce in this lab: *sets* and *tuples*.

## 1.1 Sets

If you're familiar with the mathematical concept of a set, then the Python version is just like that. If you're not, then the two main things you need to know for now are that, unlike lists, sets cannot have multiple elements, and the elements within them are not ordered. For example, `set([1,1,1,1,2,3])` returns the set $\{1, 2, 3\}$. You can always turn a list into a set using this notation (so running `set(mylist)`).

In many ways, sets resemble regular lists. You can identify the number of elements in a set `s` using `len(s)`, you can test whether or not an element `x` is in the set using `x in s` and `x not in s` respectively, and you can add and remove elements from the set using `s.add(x)` and `s.remove(x)`.

Unlike lists, sets are specifically designed to represent their associated mathematical object, so they come with additional built-in functions designed to support this. These include:

- `s1.union(s2,...,sn)` takes in an arbitrary number of sets $s_2, \ldots, s_n$ as input and outputs the *union* of these sets; i.e., the set containing all of their elements. This can also be achieved by running `s1 | s2 | ... | sn`. Mathematically, it is written as $s_1 \cup s_2 \cup \cdots \cup s_n$ (where $\cup$ is the union operator). For example, `set([1,2,3]) | set([3,4,5])` is `set([1,2,3,4,5])`.

- `s1.intersection(s2,...,sn)` takes in an arbitrary number of sets $s_2, \ldots, s_n$ as input and outputs the *intersection* of these sets; i.e., the set containing elements that are contained within all of the sets. This can also be achieved by running `s1 & s2 & ... & sn`. Mathematically, it is written as $s_1 \cap s_2 \cap \cdots \cap s_n$ (where $\cap$ is the intersection operator). For example, `set([1,2,3]) & set([3,4,5])` is `set([3])`.

- `s1.difference(s2)` takes in two sets and returns the *difference*; i.e., the set consisting of elements in the first set that are not in the second. This can also be achieved by running `s1 - s2`. Mathematically, it is written as $s_1 \setminus s_2$ (and is also sometimes referred to as *set minus*). For example, `set([1,2,3]) - set([3,4,5])` is `set([1,2])`.

- `s1.issubset(s2)` outputs a boolean indicating whether or not $s_1$ is a subset of $s_2$; i.e., whether or not all elements in $s_1$ are contained in $s_2$. Mathematically, this is written as $s_1 \subset s_2$ (if $s_2$ is strictly larger than $s_1$) or $s_1 \subseteq s_2$ (if it is possible that $s_1$ is equal to $s_2$). This can also be achieved by running `s1 < s2` (to check if $s_1 \subset s_2$) or `s1 <= s2` (to check if $s_1 \subseteq s_2$). For example, `set([1,2,3]) < set([3,4,5])` is `False`, but `set([1,2]) < set([1,2,3])` is `True`.

- `s1.issuperset(s2)` outputs a boolean indicating whether or not $s_1$ is a superset of $s_2$; i.e., whether or not $s_2 \subseteq s_1$. This can also be achieved by running `s1 > s2` (to check if $s_2 \subset s_1$) or `s1 >= s2` (to check if $s_2 \subseteq s_1$).

- `s1.isdisjoint(s2)` outputs a boolean indicating whether or not $s_1$ and $s_2$ are *disjoint*; i.e., whether or not they have no overlapping elements. Mathematically, if the sets are disjoint we write $s_1 \cap s_2 = \emptyset$ (where $\emptyset$ denotes the empty set).

## 1.2 Tuples

Tuples in Python look like lists, but with two differences: they are delimited by parentheses rather than square brackets (so `(1,2,3)` rather than `[1,2,3]`), and if they contain a single element they must put a comma after that element anyway, so you write `(1,)` rather than `(1)` (which Python will treat as the integer 1) or `[1]` (which is valid syntax for a list).

Aside from syntax, what are the actual differences? The main difference is the same as the difference between strings and lists: while lists are mutable, meaning elements can be added and removed from them, tuples are immutable. In particular, this means they do not support functions that we've seen like `append` and `remove`. On the other hand, their immutability means they can be used as keys in dictionaries, and they are also more efficient than lists. Tuples are also designed to hold objects of many different types, as we'll see in the next section.

## 1.3 Iterating over iterables

We've already seen one main way to iterate over objects like lists and strings, which is using a `for` loop. Occasionally, however, `for` loops do not provide the exact functionality we want. For example, if we need not only the entries in the list but also their indices (as we did in `dna.py`), we saw that we ended up iterating over `range`, as in:

```
>>> mystr = 'abc'
>>> for i in range(0,len(mystr)):
...     print i, mystr[i]
...
0 a
1 b
2 c
```

It turns out that we can easily iterate over both an iterable type and its indices using the built-in function `enumerate`. This function takes in an iterable type and outputs an enumerate object. While this object has no direct representation (for reasons of efficiency), we can think of it as a list of tuples, where the first item in the tuple is the index of the second item in the tuple in the original input (thus demonstrating that tuples are designed to hold objects of different types). For example:

```
>>> for (i,x) in enumerate('abc'):
...     print i, x
...
0 a
1 b
2 c
```

In addition to being arguably more elegant than the solution using `range`, this is also significantly more efficient, so is important when iterating over large datasets.

The other main reason we may end up being tempted to use `range` is if we are iterating over two lists at the same time. Consider, for example, the case in which we want to represent colors using their RGB values, so we have one master list `rgb = ['red', 'green', 'blue']`, and then individual lists like `red=[255, 0, 0]` that represent distinct colors. (Every color can be uniquely represented by having an RGB value between 0 and 255.) In this case, to capture the mapping between colors and their values, we may end up writing something like:

```
>>> for i in range(0,len(rgb)):
...     print rgb[i], color[i]
...
red 255
green 0
blue 0
```

This too can be done more efficiently using the built-in `zip` function. Like `enumerate`, `zip` creates pairs, but this time the first element in the pair is the corresponding element in the first list and the second element in the second list. So, we can do the above in a more efficient way using:

```
>>> for (col,val) in zip(rgb,red):
...     print col, val
...
red 255
green 0
blue 0
```

Basically, anytime you are iterating over `range`, it is a good idea to ask yourself if you could be using something else instead. While it doesn't matter over smaller datasets, it starts to make a big difference over larger datasets, and is also just considered better coding style.

In the above example, of course, you hopefully asked yourselves why you were using lists at all. Since the goal was to map RGB colors to their values, a dictionary is a much better fit here; for example, we could represent red as `red = {'red' : 255, 'green' : 0, 'blue' : 0}`. Here again, there is a more efficient way to iterate over both the keys and values: rather than using

```
>>> for key in red.keys():
...     print key, red[key]
...
blue 0
green 0
red 255
```

we can instead use the `items()` function to write

```
>>> for (k,v) in red.items():
...     print k, v
...
blue 0
green 0
red 255
```

## 2 Lambda, or Anonymous Functions

While Lab 04 should have convinced you that functions are an important part of programming in Python, sometimes it is really not necessary to define a function. For example, if we want to add one to a number once in a long program, then it would be silly to define a function (as we did in Lab 04) `addone` and then call `addone(x)`, as opposed to just running `x+1`. In general, the main purpose of defining functions is to reduce code duplication, so if there is a function we want to use only once, we should be able to do so succinctly without having to define it.

This brings us to `lambda` (which if you're curious is my favorite thing about Python). Just like a regular function, a `lambda` expression is supplied inputs and, by executing the code specified in a body, produces outputs. What is the difference then with regular functions? It turns out there are many,[1] but for the purposes of this module we'll highlight just a few:

- First, functions defined by `lambda` are *anonymous*, meaning they do not have names; this also means they do not get added to the environment.
- Second, `lambda` functions must have a return value, so we cannot use them to do things like append items to lists.
- Third, because they are designed to deal only with simple functions, `lambda` expressions cannot contain things like loops, or even complicated conditional statements (they can use the ternary operator `a if b else c`, however, that we learned in Week 03).

Finally, the syntax of a `lambda` expression is different to the syntax of a function definition. In particular, it looks as follows: first the keyword `lambda` is used, followed by a space and then a comma-separated list of the inputs to the function. A colon is then used to denote the body of the function, which as usual specifies the actions to take on the inputs. Crucially, this is done without using the `return` keyword. The anonymous function to add one to a number, for example, would be

```
lambda x : x + 1
```

It is even possible to define named functions using `lambda`, by doing something like

```
addone = lambda x : x + 1
```

but this negates the whole point of using `lambda` so is strongly discouraged. Even though `lambda` does not use the `return` keyword, it still does return a value (unlike a defined function that would return `None`.)

## 3 Higher-Order Functions

Thus far, we have mainly seen and defined functions that act on other data types: integers, lists, strings, etc. There is no reason, however, that a function can't also take as input another function; functions that do this are referred to as *higher-order functions*, or are said to exemplify a *functional programming* perspective. In this section, we explore some powerful built-in functions that do exactly this: `filter`, `map`, and `reduce`.

[1] `https://en.wikipedia.org/wiki/Anonymous_function`

## 3.1 Filter

Suppose that, given an iterable type, we want to *filter* out all elements in it that have a certain property; e.g., we want to know crimes that occurred only on a certain date, or the responses of only female participants in a survey.

**Task 1.** First, create a file `filter.py`; you'll add all the code you write in this subsection into this file. *Using a loop*, write and test a function `loop_filter` that, given a numeric list and a number $n$, returns the numbers in the list that are greater than $n$. For example, `loop_filter([1,2,7,8],5)` should return `[7,8]`.

While the body for `loop_filter` likely isn't too long, it turns out it's more cumbersome than is necessary. In particular, using the `filter` function, we can achieve the same goal using fewer lines of code (and, more importantly, in a more efficient way).

The syntax for this function is `filter(myfunc,myiter)`, which outputs an iterable type (typically the same type it is given as input, so if it is given a string it outputs a string). When evaluated, it applies the boolean function `myfunc` to each element in `myiter` and includes it in the output only if `myfunc` returns `True`. While `myfunc` could be a pre-defined function, this type of application is exactly where `lambda` becomes useful. For example,

```
>>> filter(lambda x : isinstance(x,str), ('a',1,1.5,'',[]))
('a','')
```

since these are the entries in the input tuple that are instances of strings (and thus for which the function `isinstance` returns `True`).

**Task 2.** Using `lambda` and `filter`, write and test a *one-line* function `lambda_filter` that, given a numeric list and a number $n$, returns the numbers in the list that are greater than $n$.

## 3.2 Map

When filtering elements from a list, we implicitly run a boolean function on every element in the list (checking, for example, if it is greater than $n$), but we never see the output of that function; instead, we just see the filtered elements for which the function evaluated to `True`.

A related function, `map`, allows us to see this. For example, we could run the following:

```
>>> mylist = [1,2,3,7,8,9]
>>> map(lambda x : x > 5, mylist)
[False, False, False, True, True, True]
```

This allows us to understand how `filter` works, but `map` is much more powerful, as we can also incorporate non-boolean functions. For example, if we wanted to add one to every element in a list, we could run:

```
>>> mylist = [1,2,3,7,8,9]
>>> map(lambda x : x + 1, mylist)
[2, 3, 4, 8, 9, 10]
```

We can also use `map` to apply previously defined functions, as in:

```
>>> map(sum,[[1,6],[2,5][3,4]])
[7, 7, 7]
```

**Task 3.** In a file `map.py`, define a function `fahrenheit` that, given a temperature in Celsius, returns the temperature in Fahrenheit (you can use the Internet or see the handout for Lab 01 for a reminder on this formula). Now, create a list representing a week's worth of temperatures and use `map` to convert these temperatures into Fahrenheit. Print out this new list.

### 3.3 Reduce

Finally, `reduce` collapses the list into a single value by repeatedly calling a binary function; i.e., it first runs the binary function on the first two elements in the list, then runs it on the result and the third element in the list, and continues until it incorporates the last item in the list. For example, to sum all of the elements in a list, you could write a function `my_sum` as follows:

```
def my_sum(mylist):
    return reduce(lambda x,y : x+y, mylist)
```

This function should have the exact same functionality as the built-in function `sum`.

### 3.4 List comprehensions

List comprehensions are a Python-specific construct that behave almost identically to `map`, but are often considered easier to read and save us from having to use and read `lambda` expressions.

In a list comprehension, we can perform operations on values in a list (or in fact, in values across multiple lists), just as we did with `map`. The syntax for doing so, for a function `myfunc` and a list `mylist`, is

```
[myfunc(x) for x in mylist]
```

Why use list comprehensions? One not very good reason is that in Python 2.7 they are (marginally) faster when `lambda` needs to be used for `map` but not for the list comprehension. (In Python 3 `map` is significantly more efficient in terms of memory usage.) The main reason they are preferred, however, is readability; e.g., consider the following examples:

```
>>> [x + 1 for x in [1,2,3]]
[2, 3, 4]
>>> map(lambda x : x+1, [1,2,3])
[2, 3, 4]
```

List comprehensions are also arguably easier when handling multiple lists, or conditional statements. For example, consider:

```
>>> list1 = [1,2,3,4,5]
>>> [x ** 2 for x in list1 if x != 2]
[1, 9, 16, 25]
>>> map(lambda x : x ** 2, filter(lambda x : x != 2, list1))
[1, 9, 16, 25]
```

You have to be careful to understand how they work, however, as in:

```
>>> list1 = [1,2,3]
>>> list2 = [4,5]
>>> [(x+1,y) for x in list1 for y in list2]
[(2, 4), (2, 5), (3, 4), (3, 5), (4, 4), (4, 5)]
```

Ultimately, deciding between list comprehensions and `map` is largely a personal decision, and for the purposes of this module both should work equally well.

## 4 Putting It All Together

To work with these higher-order functions, we'll continue with the metal theft dataset we worked with in Lab 04, which can still be found at `data/church_metal_theft.csv`. Please see the handout for Lab 04 if you need a reminder on the structure of this dataset.

In Lab 04, we interacted only with the timing of the documented crimes. In this lab, we'll focus on the free text field and the details it conveys about the nature of the theft.

The two most common types of metal stolen from churches are lead and copper. Within these categories, there are several specific types of objects that are often stolen. For lead, the two most common things stolen are lead flashing (i.e., strips of lead around a roof to provide a water barrier) and lead roofing. For copper, the most common thing stolen is the lightning rod on the roof.

**Task 4.** First, either copy over your `church_theft.py` file from last week, or use my solution.

Next, using `filter` and `map` and the free text field in the dataset, identify the thefts that involve lead and the ones that involve copper. How many total crimes are there for each? Print out the answer for both lead and copper in your code. (Hint: Keep in mind that some of the text fields are in all caps, so you might want to use the `lower` function, which is run as `mystr.lower()` and returns `mystr` as a string all in lowercase.)

Next, using keywords associated with the specific items stolen, refine these results further. How many crimes involve the theft of lead flashing, and how many involve the theft of parts of a lead roof? How many lead-based crimes involve neither of these specific items? Print out the answers to each of these questions in your code. (Hint: You will likely find it useful to use sets to answer the last one.)

For the crimes that you believe involve stealing parts of a roof, manually take a look at a few (or at all) by printing them out. Do you see anything either in terms of the number of items related to roofing or in your manual search of these items that is problematic? Please write down just a sentence or two to answer this, either as text in a separate file `church_theft.txt` or as an extended comment in your code.

Finally, how many of the copper thefts involve the theft of the lightning rod? Here you may find it useful to know that people often misspell the word lightning (it is up to you to identify how, again by doing a small manual inspection), and that they also refer to it as a lightning conductor. Again, print out the answer to this question in your code.

# 5 Handing In

As always, feel free to continue trying out the various concepts we've covered in this lab. Once you are done, add all relevant files into a `lab05` directory; this should mean adding:

- `filter.py`
- `map.py`
- `church_theft.py`
- `church_theft.txt` (optional)

Push these files, along with the usual `partner.txt`, to the remote repository, and you're done! Please check with one of us on your way out.