Skillful Python Iterables

Eh 301 User Help Manual

Mitchell T. Dodson

1. Introduction

Python is a high-level dynamically typed object-oriented programming language rich with built-in functions as well as what its creators refer to as "syntactic sugar," or small and convenient elements of the language's syntax that make solutions common programming problems easy and readable. Luxurious as these features are in implementation, to a beginner they often seem abstract and unconventional.

This manual intends to provide an introduction to several built-in functions and pieces of syntactic sugar which pertain to iterable objects such as lists, sets, and tuples. These skills enable the user to create, mutate, sort, and combine iterables clearly and easily.

The manual assumes that you have a basic understanding of object-oriented programming and the concept of functions and function references. If these concepts are unfamiliar to you, take a look at one of many free online resources such as W3SCHOOLS.COM, or check out a relevant book such as O'Reilly's *Head-First Object-Oriented Analysis & Design*

2. Background and Basics

2.1 what is an iterable?

Iterables are objects in Python that form a collection of other objects. In particular, "iterable" referes to a type of object that allows for the notion of going to the "next" or "previous" item, in turn enabling you to *iterate* through the structure via looping and other methods.

In Python, standard iterables include (among others) lists, sets, and tuples, each of which have unique qualities. The most generalized Python iterable, *lists*, are most conceptually similar to a linked list data structure. Lists are capable of containing any type and any number of objects, including repeat items, and allow you to dynamically append and remove objects at will.

Like lists, *tuples* can contain any number of objects including repeat items, but items cannot be added, removed, or changed after the tuple is created (though the tuple can be converted to any other iterable type). These qualities may seem useless at first, but tuples are more memory-efficient, and allow you to more safely pass iterables to and from functions.

Finally, *sets* are similar to lists in that they allow you to add, remove, or change objects at will, and to store any number of objects within, however unlike tuples and lists, sets don't allow for repeated items. As an example, if the integer **2** is already included in a set, trying to add **2** to the set will not change the set at all.

Note that the iterable type of a data structure can be changed at will by passing the structure into the identity function of the desired type. For example, to convert a tuple `my_tuple` to a set with `set(my_tuple)`, or a list with `list(my_tuple)`

2.2 python basics

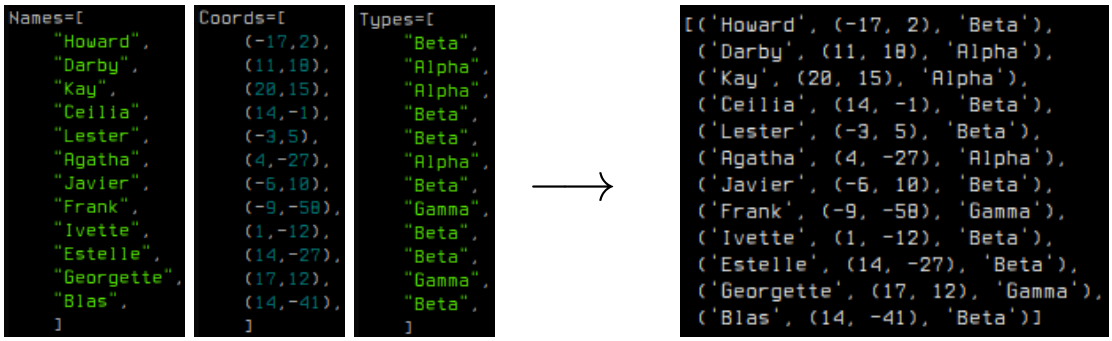
Unlike C, C++, or any other language that relies on dynamically-allocated memory, Python allows you to include any size and kind of object in an iterable structure with any other size and kind of object. As an example, a Python list object is happy containing 3 strings, 1 integer, even other iterables (see the tuple included in `my_list`, Figure 1).

```
my_list = [
    "any random string",
    42,
    ("a tuple", 17, "with 3 items"),
    lambda a: a[::-1],
    ]
```

Figure 1: type-agnostic iterables

Furthermore, one of Python's core philosophies asserts that *everything* is an object. This includes functions as well as static elements of data like integers or strings. The implications of this design are legion, some of which will be covered later in this manual, however for now simply bear in mind that functions can be assigned variable names or even included in iterables and other structures (see lambda, aka anonymous function included in Figure 1. If this is unfamiliar, don't worry; lambda functions are covered in Section 5).

3. Zip



INPUT OUTPUT

When you have multiple iterables, all of which contain the same number of items, you can combine the iterables into a single iterable with items corresponding to the same indices nested in tuples therein by utilizing the `zip` built-in function.

```
hurricanes = list(zip(Names,Coords,Types))
```

Figure 2: zip implementation

As an example, the above input iterables, `Names`, `Coords`, and `Types` each have 12 data elements pertaining to 12 individual hurricanes such that the hurricane name at index n corresponds to the coordinates at index n and hurricane type at index n

To use the 'zip' function:

1. Verify that each input iterable has the same number of items.
2. Pass each iterable to the `zip()` function as a positional argument. Note that order matters here; the items of the iterable passed as first argument will be the first items to appear in the resulting tuple.
3. The object returned by `zip()` is a function acts like an iterator, which cannot be parsed like a regular iterable. In order to turn it into a usable form, pass the function into an identity function as discussed in Section 2.1. In the above example, we make our new collection of nested tuples into a list.

After successfully using the `zip()` built-in function, you should be left with a single new iterable of the type you chose containing tuples. Every tuple contained therein will contain one item from each input iterable corresponding to the same index as the new tuple. Furthermore, the order of appearance of items in the original iterables is conserved in the final product. If you want to change the type of the nested tuples, see `map` function (Section 4).

4. Map

INPUT OUTPUT

Python's `map()` function gives you the opportunity to apply a function to every member of an iterable. As with many of the iterable methods covered in this manual, this is quicker, more readable, and more memory-conservative than looping manually through each member of the iterable.

```
hurricanes = list(map(
    list,
    hurricanes,
    ))
```

Figure 3: map implementation

Recall that *everything* in Python is an object that can be passed to and called by other objects. This principle allows you to pass a function to the `map()` function *as a positional argument*.

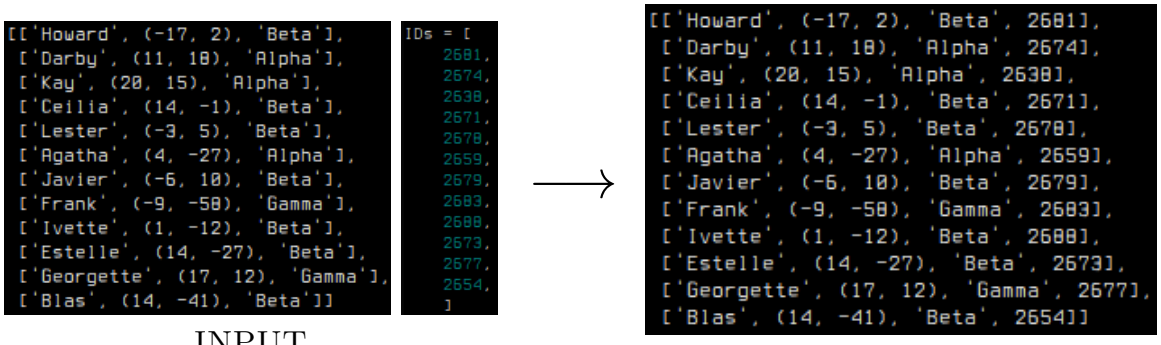
In the provided example, we use the identity function for lists as the function to apply to each item in `hurricanes`. This will convert each tuple initially nested in `hurricanes` to a nested list.

To use the 'map' function:

1. Choose a function to apply individually to each item in an iterable. You may need to assign your function to a variable; otherwise, use an anonymous function (Section 5).
2. Pass `map()` the function as the first argument, and the subject iterable as the second.
3. Like the `zip()` function, `map()` returns an iterable-like object that cannot be parsed like an iterable. Simply use the identity function of the desired iterable type to turn the `map()` object into a usable form.

After successfully using the `map()` function, you will have a new iterable of the selected type with the same number of elements, such that each item will be the result of the provided function applied to item originally at that index.

5. Lambda



INPUT OUTPUT

Python's "everything is an object" design philosophy, which allows you to pass functions to other functions (as seen in Section 4) renders functions far more dynamic and accessible than in other languages. Often you will not want to create an entire function definition using `def(*args, **kwargs)` for simple functions or functions meant to be passed as arguments to other functions. Python's `lambda` method (a.k.a *anonymous functions*) remedy this by enabling you to create a one-line function with any number of arguments.

```
hurricanes = list(map(
    lambda a,b: a+[b],
    hurricanes,
    IDs,
    ))
```

Figure 4: lambda implementation

Lambdas are refered to as anonymous functions because they don't require you to assign them to a variable; instead, you can define lambdas within the argument literal of whatever function you are passing them to (see Figure 4).

To use the 'lambda' function:

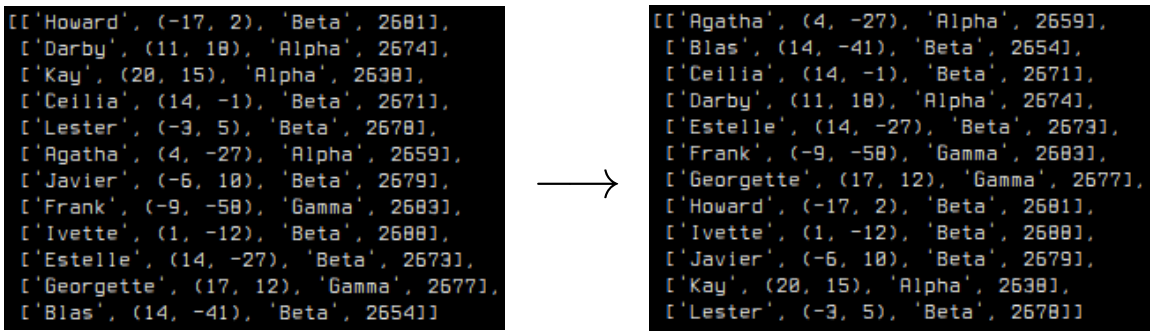
1. After the `lambda` keyword, name positional arguments separated by commas and followed by a colon (variables correspond to the order of appearance of arguments).

2. After the colon, define the function's operations as you would any other function using the variables declared before the colon. These arguments are restricted to a single line.

3. If you lambda is intended as be an argument to another function, simply place the lambda definition in the argument literal of the parent function. Otherwise, assign the lambda to a variable name as you would any other variable. You can pass arguments to this variable as you would a regular function.

After successfully declaring a `lambda` function, you will have a function object that can be passed as an argument to other functions or assigned to any number of variables.

6. Sorted



INPUT OUTPUT

Python's `sorted()` built-in function allows you to sort an iterable based on the alphabetical or numerical value of a specified key. This method is particularly useful when you have an unsorted iterable or an iterable initially sorted by a contained value other than the desired subject element.

```
hurricanes = list(sorted(
    hurricanes,
    key=lambda a: a[0],
    ))
```

Figure 5: sorted implementation

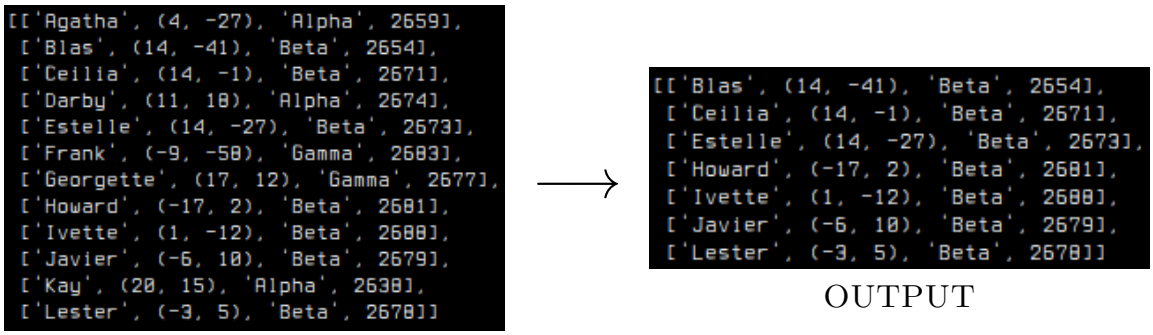
As with `map()` and many other functions, one of the keyword arguments of `sorted()` is a function specifying which element of the iterable is the 'sortable' part. In the above example, we want to sort the `hurricanes` list alphabeticlaly by the hurricane name, which is the first element of the lists nested within.

To use the 'sorted' function:

1. Provide `sorted` an iterable object to be sorted as the first positional argument.
2. If the first-level items in the iterable aren't the items you want to sort by, provide a function as a keyword argument with specifier "`key=`" that parses the item with which you want to sort the iterable.
3. `Sorted` returns an iterator function object that must be converted to the desired iterable by wrapping with an identity function. In the above example, we turn the new iterable into a list using the `list()` identity function.

If used properly, the `sorted()` function will return a new iterable of the specified type containing the same elements value-sorted according to the proveded key.

7. Filter



INPUT OUTPUT

Python's `filter()` built-in function allows you to reduce a provided iterable to a new iterable by applying a function to each element. If the given function evaluates '`True`' (returns anything other than `False` or `None`) when applied to an element in the original iterable, the element will be included in the new iterable returned by the function.

```
hurricanes = list(filter(
    lambda a: a[2]=="Beta",
    hurricanes,
    ))
```

Figure 6: filter implementation

`filter()` functions are particularly useful when you have a large iterable containing some elements that are relevant to future operations and others which aren't. If a single operation can determine the relevance of an element (such as checking for equivalency in the example), use an anonymous function as the filter condition.

To use the 'filter' function:

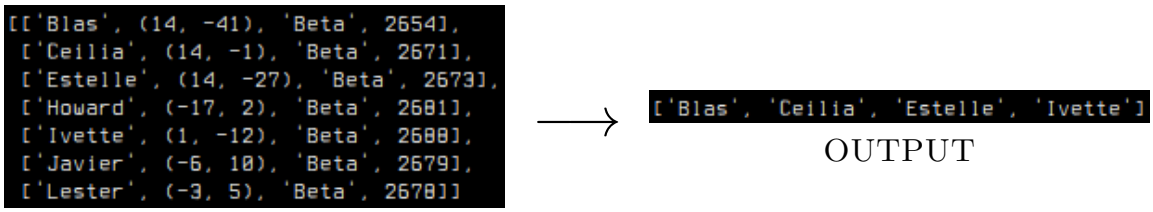
1. Provide either a function assigned to a variable or an anonymous function as the first positional argument. This function should evaluate '`True`' for any items you want to include in the resulting iterable.

2. Provide an iterable to filter as the second positional argument. Be sure that this iterable's types are appropriate for the condition-checking function.

3. `filter()` returns an iterable-like function object; in order to convert this object to something usable as an iterable, wrap the `filter` function with the identity function of the desired iterable type.

If successful, this process will provide a new iterable with the same number or fewer items than the original iterable; each item will be an item that results in a '`True`' value when passed through the provided function.

8. Comprehension



Unlike the other Python built-in methods covered in this manual, Python comprehensions – the paragon of syntactic sugar – don't take the form of a function in the conventional sense, in that you don't pass objects and functions to a comprehension as an argument. Instead, comprehensions provide a template for defining functions and providing inputs *as a single argument*. This contained function must follow a specific (but still incredibly customizable) format of statements.

```
N_hc = [ i[0] for i in hurricanes if (i[1][0] > 0) ]
```

Figure 7: comprehension implementation

To use a comprehension:

1. Wrap everything that follows in square brackets `[]` to call the comprehension method.
2. Define `[ x for x in y ]` where `x` is an arbitrary variable name and `y` is the name of a predefined iterable from which the new iterable will be derived. In this case, `x` identifies the first-level elements contained within `y`.
3. If you want the items included in the new iterable to be a modified version, a derived object, or a child element of the first-level items in the parent iterable, use any inline argument to specify the item that you ultimately want to include. In the above example, the first-level items from `hurricanes` are the 3-member lists assigned to `i`. Since we only want the *names* of the hurricanes, we call the 0'th index of `i`.
4. If you want to filter included items based on any number of conditions, use a series of conditional statements such as `if`, `else`, `not`, et cetera. Like the `filter` bultin (Section 7), the item will be included if and only if all the provided statements evaluate `True`.

If successful, your comprehension will return a list of all the derived items that pass the given conditions. If a iterable type other than list is desired, simply convert the product using an identity function. Truthfully, this manual only scratches the surface of what comprehensions are capable of; the function is stunningly versatile, so some element of trial and error is healthy for determining the extremes of comprehensions' use cases.