# Appendix of Useful Functions

## 0.1 Tuples

*Length*: The function `len` returns the length of a tuple. For example, `len((3, 4, 5))` evaluates to 3.

*Item Selection*: The expression `tup[pos]` evaluates to the item at position `pos` of tuple `tup`. For example, if `tup` is `(1, 2, 3, 4)`, then `tup[0]` evaluates to `1` and `tup[2]` evaluates to `3`.

*Slicing*: The expression `tup[start:end:size]` evaluates to a smaller tuple that contains the items from `start` to `end` (excluding the `end`), with a step size of `size`. All arguments are optional. For example, if the tuple `tup` is `(1, 2, 5, 7, 8, 10)`, then `tup[0:3]` returns the tuple `(1, 2, 5)` and `tup[0:4:2]` returns the tuple `(1, 5)`.

*Concatenation*: Tuples can be concatenated into a larger tuple, which contains the elements of both tuples, using the `+` operator. For example, `(0, 1, 2) + (4, 5)` yields the tuple `(0, 1, 2, 4, 5)`.

## 0.2 Ranges

The expression `range(start, end, size)` returns an iterable "range object" (*not* a tuple), which is a sequence that contains the integers from `start` to `end` (excluding the `end`), with a step size of `size`. The third argument is optional. If the first and third arguments are not provided, the start value is assumed to be 0. For example,

```
for val in range(0, 5):
    print(val)
```

will print the values `0`, `1`, `2`, `3` and `4`.

## 0.3 Map, Filter, Reduce

`map(func, seq)` applies the function `func` to the items in the sequence `seq` and returns the results in an iterable "map object" (*not* a tuple). You would use the `tuple` constructor to create a tuple of these results. For example, `tuple(map(lambda x: x*x, (1, 2, 3, 4)))` applies the squaring function to all of the items in the sequence (here, a tuple), and returns the tuple `(1, 4, 9, 16)`.

`map` can also take in more than one sequence, in which case the function `func` is applied first to the first elements of all sequences, then to the second elements of all sequences, and so on. For example, `tuple(map(lambda x, y, z: x+y+z, (1, 2, 3), (4, 5, 6), (7, 8, 9)))` returns the tuple `(1+4+7, 2+5+8, 3+6+9)` or `(12, 15, 18)`.

`filter(pred, seq)` filters out the items in the sequence `seq` that do not satisfy the predicate `pred` and returns the remaining items in an iterable "filter object" (*not* a tuple). You would use the `tuple` constructor to create a tuple of these items. For example, `tuple(filter(lambda x: x>5, (3, 5, 7, 9, 13)))` filters out all of the items in the sequence (here, a tuple) that are not greater than 5, returning the tuple `(7, 9, 13)`.

`reduce(func, seq, initial_value)` returns the result of applying `func` cumulatively to the items of a sequence `seq` from left to right, starting with the `initial_value`. For example, `reduce(lambda x, y: x+y, (1, 2, 3, 4), 10)` returns the value `(((((10 + 1) + 2) + 3) + 4)`. The argument `x` maintains the value accumulated so far, and the argument `y` will be iterated through the elements of the sequence from left to right.

If `initial_value` is not provided, `reduce` will use the first element of the sequence as the initial value; if the sequence is empty, and no initial value is provided, `reduce` will return an error.

### 0.4 Immutable Recursive Lists (IRLists)

`make_irlist(first, rest=empty_irlist)`: Returns a new IRList by prepending item `first` to the front of the IRList `rest`, which is the empty IRList by default.

`irlist_first(irlist)`: Returns the first item in the IRList `irlist`.

`irlist_rest(irlist)`: Returns an IRList of all items *except* the first item in the IRList `irlist`.

`could_be_irlist(thing)`: Returns `True` if `thing` could be an IRList, and `False` otherwise.

`irlist_populate(*items)`: Returns a new IRList populated with the items provided in `items`.

`irlist_len(irlist)`: Returns the length of the IRList `irlist`.

`irlist_select(irl, pos)`: Returns item at position `pos` (counting from zero) of the IRList `irl`.

`irlist_insert(irlist, index, item)`: Returns a new IRList where the item at position `index` (counting from zero) is replaced with the new item `item`.

`irlist_remove(irlist, index)`: Returns a new IRList where the item at position `index` (counting from zero) of IRList `irlist` is removed.

`irlist_append(irlist1, irlist2)`: Returns a new IRList that is the result of adding the elements in IRList `irlist2` to the end of IRList `irlist1`.

`irlist_map(func, irlist)`: Returns a new IRList that contains the results of applying `func` to every item in IRList `irlist`.

`irlist_filter(pred, irlist)`: Returns a new IRList that contains the items in IRList `irlist` that satisfy the predicate `pred`.

`irlist_str(irlist)`: Returns a string representation of the IRList `irlist`.

### 0.5 Immutable Dictionaries (IDicts)

`make_idict(*mappings)`: Returns a new IDict from the tuple of key-value pairs provided in `mappings`.

`idict_keys(idict)`: Returns a tuple of the keys belonging to the IDict `idict`.

`idict_values(idict)`: Returns a tuple of the values that are mapped to in the IDict `idict`.

`could_be_idict(thing)`: Returns `True` if `thing` could be an IDict, and `False` otherwise.

`idict_select(idict, key)`: Returns the value that the key `key` maps to in the IDict `idict`, or `None` if the key is not present in `idict`.

`idict_insert(idict, key, value)`: Returns a copy of the IDict `idict`, updated with the key `key` now mapped to value `value`.

`idict_len(idict)`: Returns the number of mappings in the IDict `idict`.

`idict_items(idict)`: Returns a tuple of key-value pairs stored in IDict `idict`.

`idict_str(idict)`: Returns a string representation of the IDict `idict`.