16.9 Generalized Sorting

So far in this chapter, we've studied several algorithms for sorting data. While these algorithms vary greatly in their approaches and running times, our implementations so far have all had one thing in common: they sort elements of a collection by comparing the *values* of each element.

But in real-world applications, we usually deal with collections of compound data, and sort them by some attribute or computed property of element. For example:

- sort strings by their length
- sort collections of integers by their sum
- sort `Person` objects by their `age` attribute

In this section, we'll learn how to generalize the sorting algorithms we've studied so far to enable sorting based on arbitrary computations performed on the elements. Along the way, we'll learn about a new technial feature of the Python programming language which makes it surprisingly straightforward to support this generalized sorting.

A first example: sorting by string length

To start, consider the following function specification:

```
def sort_by_len(lst: list[str]) -> None:
    """Sort the given list of strings by their length.

    >>> lst = ['david', 'is', 'cool', 'indeed']
    >>> sort_by_len(lst)
    >>> lst
    ['is', 'cool', 'david', 'indeed']
    """
```

Since we know sorting is involved, we could try to reuse one of our existing sorting algorithms, like `insertion_sort`:[1]

[1] We'll stick with insertion sort as our running example for this section, but everything we say applies just as well to the other sorting algorithms in this chapter.

```
def sort_by_len(lst: list[str]) -> None:
    """Sort the given list of strings by their length.

    >>> lst = ['david', 'is', 'cool', 'indeed']
```

```
    >>> sort_by_len(lst)
    >>> lst
    ['is', 'cool', 'david', 'indeed']
    """
    insertion_sort(lst)  # This is tempting, but doesn't quite work!


# Insertion sort implementation, for reference
def insertion_sort(lst: list) -> None:
    for i in range(0, len(lst)):
        _insert(lst, i)


def _insert(lst: list, i: int) -> None:
    for j in range(i, 0, -1):
        if lst[j - 1] <= lst[j]:
            return
        else:
            lst[j - 1], lst[j] = lst[j], lst[j - 1]
```

Of course, this doesn't work! If we call this function, `lst` will be sorted alphabetically, using Python's default behaviour when comparing strings with `<=`. Perhaps surprisingly, this behaviour all comes down to a single line of code: the `if` condition `lst[j - 1] <= lst[j]`, which is the only place in this algorithm that actually compares the elements of `lst`.

So we can achieve our desired behaviour by modifying this comparison to check `len(lst[j - 1]) <= len(lst[j])` instead. The following is a correct implementation of `sort_by_len`:

```
def sort_by_len(lst: list[str]) -> None:
    """Sort the given list of strings by their length.

    >>> lst = ['david', 'is', 'cool', 'indeed']
    >>> sort_by_len(lst)
    >>> lst
    ['is', 'cool', 'david', 'indeed']
    """
    insertion_sort_by_len(lst)


# Insertion sort implementation, for reference
def insertion_sort_by_len(lst: list) -> None:
    for i in range(0, len(lst)):
        _insert_by_len(lst, i)


def _insert_by_len(lst: list, i: int) -> None:
    for j in range(i, 0, -1):
        if len(lst[j - 1]) <= len(lst[j]):  # This line has changed!
            return
```

```
            else:
                lst[j - 1], lst[j] = lst[j], lst[j - 1]
```

But while correct, this implementation is unsatisfying: we don't want to have to write a separate function for each case of custom sorting. While this example is a good start, we need to do better to truly *generalize* our sorting algorithms.

A `key` parameter

When we first introduced functions all the way back in Section 2.1, we said that they were a way of defining a block of code that could be reused throughout a program, with only specific parameter values changed. This is exactly what we want for generalized sorting: in `_insert`, we don't want a specific condition like `lst[j - 1] <= lst[j]` or `len(lst[j - 1]) <= len(lst[j])`, but rather a condition with a customizable comparison.

So our idea to implement generalized sorting is to introduce a new parameter `key`, which represents a *function* that specifies how to compute the values that will be compared. Here is our implementation of generalized insertion sort:

```
from typing import Callable


def insertion_sort_by_key(lst: list, key: Callable) -> None:
    """Sort the given list using the insertion sort algorithm.

    The elements are sorted by their corresponding return values when passed
        to key.
    """
    for i in range(0, len(lst)):
        _insert_by_key(lst, i, key)


def _insert_by_key(lst: list, i: int, key: Callable) -> None:
    """Move lst[i] so that lst[:i + 1] is sorted.
    """
    for j in range(i, 0, -1):
        if key(lst[j - 1]) <= key(lst[j]):  # This line has changed again!
            return
        else:
            lst[j - 1], lst[j] = lst[j], lst[j - 1]
```

Even though this looks like a simple change, the extra `key` parameter has unlocked a whole new world of possibilities for our insertion sort algorithm. Here are some examples in the Python console showing how we can pass in a *function* to customize this sorting:

```
>>> strings = ['david', 'is', 'cool', 'indeed']
>>> insertion_sort_by_key(string, len)  # Sort strings by comparing string lengths
```

```
>>> strings
['is', 'cool', 'david', 'indeed']

>>> sets_of_ints = [{1, 10}, {500}, set(), {-1, -2, -3}]
>>> insertion_sort_by_key(sets_of_ints, sum)  # Sort sets_of_ints by comparing sums
>>> sets_of_ints
[{-1, -2, -3}, set(), {1, 10}, {500}]
```

With this in hand, we can implement our original `sort_by_len` example as follows:

```
def sort_by_len(lst: list[str]) -> None:
    """Sort the given list of strings by their length.

    >>> lst = ['david', 'is', 'cool', 'indeed']
    >>> sort_by_len(lst)
    >>> lst
    ['is', 'cool', 'david', 'indeed']
    """
    insertion_sort_by_key(lst, len)
```

What's more, the `key` parameter just needs to be a function, and so we can define our functions and pass them into `insertion_sort_by_key`. Here's an example:

```
def count_a(s: str) -> int:
    """Return the number of 'a' characters in s.

    >>> count_a('david')
    1
    """
    return s.count(a)


def sort_by_a_count(lst: list[str]) -> None:
    """Sort the given list of strings by their number of 'a' characters.

    >>> lst = ['david', 'is', 'amazing']
    >>> sort_by_a_count(lst)
    >>> lst
    ['is', 'david', 'amazing']
    """
    insertion_sort_by_key(lst, count_a)
```

Built-in sorting, revisited

Since generalized sorting is extremely useful, it's probably not surprising that the built-in `sorted` and `list.sort` support it. It turns out that these two functions using the same approach we've described above, taking a `key` parameter:

```
>>> strings = ['david', 'is', 'cool', 'indeed']
>>> sorted(strings, len)  # Recall that sorted returns a new list...
['is', 'cool', 'david', 'indeed']

>>> sets_of_ints = [{1, 10}, {500}, set(), {-1, -2, -3}]
>>> sets_of_ints.sort(sum)  # ...and list.sort mutates its list object
>>> sets_of_ints
[{-1, -2, -3}, set(), {1, 10}, {500}]
```

The only difference is that these two functions make their `key` parameters *optional*, so that when no argument is passed for the parameter, the element values are compared directly (much like our original version of `insertion_sort`).[2]

[2] A good exercise for you is to modify our implementation of `insertion_sort_by_key` to make the `key` parameter optional.

Generalized sorting and anonymous functions

There is one last neat Python feature that we'll present to finish up this chapter. While our above `key` approach successfully allows us to customize sorting using built-in or user-defined functions, it is sometimes a bit cumbersome of define a separate function that's only going to be used once to customize a sort. Our example of defining a `count_a` function arguably fits into this category!

It turns out that there is a way in Python of creating simple functions within a larger expression, without needing to use the `def` keyword. Formally, an **anonymous function** is a Python expression that defines a new function without giving it a name.^{ Anonymous functions are the equivalent of literals for other data types. } The syntax for an anonymous function uses a new keyword `lambda`:

```
lambda <param> ... : <body>
```

Importantly, anonymous functions can only have an *expression* as their body; so while that expression can have lots of subexpressions, it cannot contain statements like assignment statements or loops. When an anonymous function is called, its body is evaluated and returned—no `return` keyword necessary.

Here are some examples of anonymous functions:

```
lambda x: x + 1
lambda lst1, lst2: len(lst1) * len(lst2)
```

Since anonymous functions are expressions, we can use them in places where expressions are required. For example, anonymous functions can appear on the right-hand side of an assignment statement:

```
>>> add_one = lambda x: x + 1  # add_one is now a function that adds 1
>>> add_one(10)
11
```

And coming back to our original motivation, anonymous functions can appear as arguments to a function call:

```
>>> strings = ['david', 'is', 'amazing']
>>> sorted(strings, lambda s: s.count('a'))
['is', 'david', 'amazing']
```