Sets

Allen B. Downey

September 15, 2018

Python provides another built-in type, called a set, that behaves like a collection of dictionary keys with no values. Adding elements to a set is fast; so is checking membership. And sets provide methods and operators to compute common set operations.

For example, set subtraction is available as a method called difference or as an operator, -. So the following function returns the result of substraction of d_2 from d_1:

```
def subtract(d1, d2):
    return d1 - d2
```

For example, consider the function has_duplicates, that checks whether in the list t there are duplicate elements:

```
def has_duplicates(t):
    d = set()
    for x in t:
        if x in d:
            return True
        d.add(x)
    return False
```

When an element appears for the first time, it is added to the set. If the same element appears again, the function returns True.

Alternatively, using sets, we can write the same function like this:

```
def has_duplicates(t):
    return len(set(t)) < len(t)
```

An element can only appear in a set once, so if an element in t appears more than once, the set will be smaller than t. If there are no duplicates, the set will be the same size as t. (Note that set(t) converts the list t to the set by extracting its distinct elements.)

Another example is the function which checks whether a given word (of type string) uses only available letters (also given as a string). We can write it like this:

```
def uses_only(word, available):
    return set(word) <= set(available)
```

The <= operator checks whether one set is a subset or another, including the possibility that they are equal, which is true if all the letters in word appear in available.