Before you turn this problem in, make sure everything runs as expected. First, **restart the kernel** (in the menubar, select Kernel→Restart) and then **run all cells** (in the menubar, select Cell→Run All).

Make sure you fill in any place that says `YOUR CODE HERE` or "YOUR ANSWER HERE", as well as your name and collaborators below:

In [ ]:
```
NAME = ""
COLLABORATORS = ""
```

CSE 30 Spring 2022 - Homework 6

Instructions

Please disregard the YOUR NAME and COLLABORATORS above. They are put there atomatically by the grading tool. You can find instructions on how to work on a homework on Canvas. Here is a short summary:

Submitting your work

To submit your work:

- First, click on "Runtime > Restart and run all", and check that you get no errors. This enables you to catch any error you might have introduced, and not noticed, due to your running cells out of order.
- Second, download the notebook in .ipynb format (File > Download .ipynb) and upload the .ipynb file to this form.
- This homework is due on **Thursday, 21 April 2022** by 11:59pm.

You can submit multiple times; the last submission before the deadline is the one that counts.

Homework format

For each question in this notebook, there is:

- A text description of the problem.
- One or more places where you have to insert your solution. You need to complete every place marked:

`# YOUR CODE HERE`

and you should not modify any other place.

- One or more test cells. Each cell is worth some number of points, marked at the top. You should not modify these tests cells. The tests pass if no error is printed out: when there is a statement that says, for instance:

  `assert x == 2`

  then the test passes if `x` has value 2, and fails otherwise. You can insert a `print(x)` (for this case!) somewhere if you want to debug your work; it is up to you.

Notes:

- Your code will be tested both according to the tests you can see (the `assert` statements you can see), *and* additional tests. This prevents you from hard-coding the answer to the particular questions posed. Your code should solve the *general* intended case, not hard-code the particular answer for the values used in the tests.
- **Please do not delete or add cells!** The test is autograded, and if you modify the test by adding or deleting cells, even if you re-add cells you delete, you may not receive credit.
- **Please do not import modules that are not part of the standard library.** You do not need any, and they will likely not available in the grading environment, leading your code to fail.
- **If you are inactive too long, your notebook might get disconnected from the back-end.** Your work is never lost, but you have to re-run all the cells before you continue.
- You can write out print statements in your code, to help you test/debug it. But remember: the code is graded on the basis of what it outputs or returns, not on the basis of what it prints.
- **TAs and tutors have access to this notebook,** so if you let them know you need their help, they can look at your work and give you advice.

Grading

Each cell where there are tests is worth a certain number of points. You get the points allocated to a cell only if you pass *all* the tests in the cell.

The tests in a cell include both the tests you can see, and other, similar, tests that are used for grading only. Therefore, you cannot hard-code the solutions: you really have to solve the essence of the problem, to receive the points in a cell.

Code of Conduct

- Work on the test yourself, alone.
- You can search documentation on the web, on sites such as the Python documentation sites, Stackoverflow, and similar, and you can use the results.
- You cannot share your work with others or solicit their help.

A Sock Drawer

Let us model a sock drawer. How does a sock drawer work? Well, when you do the washing, you end up with a bunch of socks, and you just put them in the drawer. So one operation is "put sock in drawer". The other thing you typically do is look for two matching socks. For simplicity, we will have only one attribute that we use to match socks: their color. So the operations, in detail, are:

- `add_sock` : adds a sock of a given color
- `get_pair` : ask if there is a pair of a given color, and if so, remove it
- `available_colors` : returns the set of colors for which there is at least a pair.

Of course, you could also devise other operations, but these seem to be the basic ones.

Here is how we plan to use the sock drawer:

```
# Creates a sock drawer and puts some socks in it.
sd = SockDrawer()
sd.add_sock("red")
sd.add_sock("green")
sd.add_sock("red")

# Gets the available colors
colors = sd.available_colors

# Picks a possible color
import random
color = random.choice(list(colors))

# And a sock of that color.
sock = sd.get_pair(color)
print("gotten sock:", sock, "of color:", color)
```

An aside: why does `available_colors` return a set and not a list? For two reasons. First, returning a set makes clear (and indeed, enforces) that every color should appear only once in the return result. Second, in sets, the order of elements is irrelevant. If we used a list, rather than a set, in any unit test suite we would have to:

- Check that every element appears only once in the list;
- sort the list of results before comparing it with the expected results, to ensure that different orders of elements in the returned list do not cause the test to fail.

In general, it is quite important to choose as types for variables or return values, a type that expresses as best as possible the intended properties of the value.

Another thing: we want to define `available_colors` so that it is a *property* rather than a method. In this way, for a sock drawer `d`, we will be able to write `d.available_colors` rather than `d.available_colors()`. It is a small difference, but in my opinion it does make the code look a little bit cleaner.

What is a good implementation? At first, one may consider simply implementing the drawer as a list of socks, where each sock is represented by its color. This is simple enough at first:

In [1]:
```
class SockDrawer(object):

    def __init__(self):
        self.drawer = []

    def add_sock(self, color):
        """Adds a sock of the given color to the drawer."""
        self.drawer.append(color)

    def get_pair(self, color):
        """Returns False if there is no pair of the given color, and True
        if there is.  In the latter case, it removes the pair from the drawer."""
        if self.drawer.count(color) >= 2:
            self.drawer.remove(color)
            self.drawer.remove(color)
            return True
        else:
            return False

    @property
    def available_colors(self):
        """Lists the colors for which we have at least two socks available."""
        colors = set()
        for el in self.drawer:
            if self.drawer.count(el) >= 2:
                colors.add(el)
        return colors
```

Let us give this a spin, with our intended use:

In [2]:
```
# Creates a sock drawer and puts some socks in it.
sd = SockDrawer()
sd.add_sock("red")
sd.add_sock("green")
sd.add_sock("red")

# Gets the available colors
colors = sd.available_colors

# Picks a possible color
import random
color = random.choice(list(colors))
```

```
# And a sock of that color.
sock = sd.get_pair(color)
print("gotten sock:", sock, "of color:", color)
```

```
gotten sock: True of color: red
```

There shouldn't be any more sock pairs available:

In [3]:
```
sd.available_colors
```

Out[3]:
```
set()
```

In the above implementation, `@property` is a *decorator*. From the practical point of view, the `@property` decorator enables us to access the available colors of a sock drawer `sd` via `sd.available_colors` rather than `sd.available_colors()`: that is, it makes what is in truth a method call look like a variable access. Big deal, you might say; all it does is it removes the need for parentheses! True, but the eye also wants its part.

In general, Python decorators are functions that take an input a function (the function just below them), and return another function; we will have occasion of seeing more of them later, and for the moment this short introduction shall suffice.

Before we proceed further, let us write a function that puts the class through some tests. We write this as a function taking the class as an argument (yes, in Python classes can be passed as arguments to functions). In this way, when we define another (perhaps better) class that implements the sock drawer, we can test it using the same tests.

In [4]:
```
#@title Let's define a testing helper.

def check_equal(x, y, msg=None):
    if x != y:
        if msg is None:
            print("Error:")
        else:
            print("Error in", msg, ":")
        print("    Your answer was:", x)
        print("    Correct answer: ", y)
    assert x == y, "%r and %r are different" % (x, y)
```

In [5]:
```
def test_drawer(c):
    """Tests a drawer class c."""
    d = c()
    d.add_sock('red')
    d.add_sock('red')
    d.add_sock('red')
    d.add_sock('green')
    check_equal(d.available_colors, {'red'})
    check_equal(d.get_pair('red'), True)
    check_equal(d.get_pair('green'), False)
    check_equal(d.get_pair('red'), False)
    d.add_sock('blue')
```

```
    d.add_sock('blue')
    d.add_sock('blue')
    d.add_sock('red')
    check_equal(d.available_colors, {'red', 'blue'})
```

In [6]:
```
test_drawer(SockDrawer)
```

I always find it a bit anticlimatic when all the tests pass, and all you get for this accomplishment is... nothing. Yes, we could define a `PatOnTheBack` exception and raise it, but that will be for another time.

Rather, let us discuss why the above implementation of a sock drawer is not great. A list keeps elements in a ... list. So if you need to count the number of times in which an element occurs, you need to traverse the list. The method count() is faster than the do-it-yourself solution `len([x for x in self.drawer if x == color])`, as it is implemented natively, but it is still slow. On top of that, once having counted the number of colors, if the chosen color appears more than once in `get_pair`, we call remove() *twice* to remove it. This is just esthetically terrible. We go over the list three times, each time unable to make use of the information we found the previous time.

In the big scheme of things, the implementation might be fine. "Going over the list" takes time proportional to the length of the list (it is a *linear-time* operation), and unless a person has a very large number of socks, the performance may never become a bottleneck. What is really disturbing, to me at least, is the ugliness of the code.

A Better Sock Drawer

Let us aim at a better solution. The problem with our list-based implementation, fundamentally, is that all our questions can be answered by knowing how many socks we have for each color, and the list implementation stores this information in unary format, and mixing all colors together. A much more rational way of remembering the status of the drawer is to associate with each color the number of socks we have. Thus, our main data structure will be a dictionary mapping colors to integers.

A short aside. If `d` is a normal dictionary, and you want to add `quantity` to `color`, you end up writing code like this:

In [7]:
```
# Just to initialize
d = {'green': 4}
color = 'red'
quantity = 2

if color in d:
    d[color] += quantity
else:
    d[color] = quantity
```

Yes, you can do a little better, as in:

In [8]:
```
d[color] = d.get(color, 0) + quantity
```

but this is not nearly as elegant as being able to access a dictionary with square-bracket notation. We would like some way of avoiding the test, or the concern, of whether a key is already in the dictionary.

defaultdicts

The `defaultdict` collection type does precisely this. A defaultdict dictionary, when accessed, returns the value associated with the key, if any, and otherwise returns a default value. You specify the default value when you create the defaultdict, by passing to its creator a *factory* function. The factory function is called any time a default value is needed, and should return the default value.

In our case, we want a dictionary where the default value is 0; so we use the function `int` which, when called without arguments, returns 0. Yes, it is a little confusing that `int` is both a type and a function...

In [9]:
```
int()
```

Out[9]:
```
0
```

So we can create a dictionary with default value 0 as follows:

In [10]:
```
from collections import defaultdict
d = defaultdict(int)
```

In [11]:
```
d
```

Out[11]:
```
defaultdict(int, {})
```

When an element is not found, rather than throwing an error, our defaultdict returns value 0:

In [12]:
```
d['red']
```

Out[12]:
```
0
```

And we can write code like this:

In [13]:
```
d['red'] += 2
print(d['red'])
print(d['green'])
```

```
2
0
```

With this we can define a better sock drawer:

In [14]:
```
class SmartDrawer(object):

    def __init__(self):
        self.socks = defaultdict(int)

    def add_sock(self, color):
        self.socks[color] += 1

    def get_pair(self, color):
        if self.socks[color] > 1:
            self.socks[color] -= 2
            return True
        else:
            return False

    @property
    def available_colors(self):
        return {c for c, n in self.socks.items() if n > 1}
```

Note how this code is not only more efficient than our previous attempt, but also more concise. Let us test this class.

In [15]:
```
test_drawer(SmartDrawer)
```

Arithmetic Dictionaries

At the core of our sock drawer implementation is a fundamental data structure which, for some reason, does not come standard in programming languages: a dictionary that has numerical values (and arbitrary keys).

A number is a number. An *arithmetic dictionary* d is a mapping from a set of keys K to numbers (let's say, to $\mathbf{R}$), that is,

$$d : K \mapsto \mathbf{R}\,.$$

In other words, d is a function from K to $\mathbf{R}$.

Functions can be combined with arithmetic operators: if you have, for instance, $f_1(x) = x + 2$, and $f_2(x) = x^2$, you can compute $g = f_1 + f_2$, where $g(x) = x^2 + x + 2$.

The same we can do with arithmetic dictionaries: if we have

$$d_1 : K \mapsto \mathbf{R} \qquad d_2 : K \mapsto \mathbf{R}$$

we can add them as $d = d_1 + d_2$, where $d[k] = d_1[k] + d_2[k]$ for all $k \in K$, and similarly for $-$, $/$, and $*$.

In fact, it will be convenient to combine also dictionaries that have different sets of keys. In that case, if a key k is missing from a dictionary d, we will take for $d[k]$ a default value that

corresponds to the *neutral elements* with respect to the operation used to combine dictionaries: 0 for $+$ and $-$, and 1 for $\times$ and /.

For example, if we have:

```
d1 = {'cat': 4, 'bird': 2}
d2 = {'cat': 5, 'dog': 3}
```

and we compute `d = d1 + d2`, we will obtain

```
d = {'cat': 9, 'bird': 2, 'dog': 3}
```

since the missing elements, such as `'bird'` in `d2`, will be assumed to have value 0.

We note that Python includes, in the `collections` module, the `Counter` class, which supports addition. Arithmetic dictionaries are a more general version of the `Counter` class.

Rather than letting you define the class `AD` on your own, for once, we provide you with some help, giving you a class that implements additon and subtraction. We will leave multiplication and division for you to implement.

The reason is this. If we left it to you, many of you would be tempted to dive right in and write something like this:

In [16]:
```
class MehAD(dict):

    def __add__(self, other):
        d = MehAD()
        other_keys = set(other.keys()) if isinstance(other, dict) else set()
        for k in set(self.keys()) | other_keys:
            d[k] = self.get(k, 0) + (other.get(k, 0) if isinstance(other, dict) els
        return d
```

In [17]:
```
d1 = MehAD()
d2 = MehAD(red=2, green=3) # dict(red=2, green=3)
d1 + d2
```

Out[17]:
```
{'green': 3, 'red': 2}
```

In [18]:
```
d2 + 3
```

Out[18]:
```
{'green': 6, 'red': 5}
```

This would work. But then, when you set to implement `__sub__`, you would have to repeat more or less the same code, except for `-` instead of `+` in line 10. And the code for `__mul__` and `__truediv__` would also be very similar.

We want to teach you how to factor the common code. The idea is to define a method `_binary_op` that takes as arguments the left operand, right operand, the function to be

used to combine the operands (addition, subtraction, etc), and the element to be used when the key is not in the dictionary. All the operands can then be defined in terms of `_binary_op`. This factorization of common code has two benefits:

- it makes the code more concise, and
- it ensures that, if you find a bug and fix it, *all* the operations will benefit from the fix.

Code factorization means gathering together the common code, eliminating repetitions; just as factoring $ab + ac$ into $a(b + c)$ removes the repetition of a. It turns out that the deep reason why code factorization is important is not conciseness: rather, it is the fact that once a bug is discovered in the factorized code, all uses of the code will benefit from it.

The code is as follows:

In [19]:
```
class AD(dict):

    def __add__(self, other):
        return AD._binary_op(self, other, lambda x, y: x + y, 0)

    def __sub__(self, other):
        return AD._binary_op(self, other, lambda x, y: x - y, 0)

    @staticmethod
    def _binary_op(left, right, op, neutral):
        r = AD()
        l_keys = set(left.keys()) if isinstance(left, dict) else set()
        r_keys = set(right.keys()) if isinstance(right, dict) else set()
        for k in l_keys | r_keys:
            # If the right (or left) element is a dictionary (or an AD),
            # we get the elements from the dictionary; else we use the right
            # or left value itself.  This implements a sort of dictionary
            # broadcasting.
            l_val = left.get(k, neutral) if isinstance(left, dict) else left
            r_val = right.get(k, neutral) if isinstance(right, dict) else right
            r[k] = op(l_val, r_val)
        return r
```

In the implementation above, we have factored together what would have been the common part of the implementation of $+$ and $-$ in the method `_binary_op`. The `_binary_op` method is a *static method:* it does not refer to a specific object. In this way, we can decide separately what to provide to it as the left and right hand sides; this will be useful later, as we shall see. To call it, we have to refer to it via `AD._binary_op` rather than `self._binary_op`, as would be the case for a normal (object) method. Furthermore, in the `_binary_op` method, we do not have access to `self`, since `self` refers to a specific object, while the method is generic, defined for the class.

The `_binary_op` method combines values from its `left` and `right` operands using operator `op`, and the default `neutral` if a value is not found. The two operands `left`

and `right` are interpreted as dictionaries if they are a subclass of `dict`, and are interpreted as constants otherwise.

We have defined our class AD as a *subclass* of the dictionary class `dict`. This enables us to apply to an AD all the methods that are defined for a dictionary, which is really quite handy.

In [20]:
```
d1 = AD()
d1['red'] = 2
print(d1)
d2 = AD({'blue': 4, 'green': 5})
print(d2)
```

```
{'red': 2}
{'blue': 4, 'green': 5}
```

In the code above, note how we can write `d1['red']`: our ability to access ADs via the square-bracket notation is due to the fact that ADs, like dictionaries, implement the `__getitem__` method. Note also that AD inherits from `dict` even the initializer methods, so that we can pass to our dictionary another dictionary, and obtain an AD that contains a *copy* of that dictionary.

We can see that addition and subtraction also work:

In [21]:
```
print(AD(red=2, green=3) + AD(red=1, blue=4))
print(AD(red=2, green=3) - AD(red=1, blue=4))
```

```
{'green': 3, 'red': 3, 'blue': 4}
{'green': 3, 'red': 1, 'blue': -4}
```

We let multiplication and division to you to implement.

In [22]:
```
### Exercise: Implement multiplication and division

def ad_mul(self, other):
    # YOUR CODE HERE
    return AD._binary_op(self, other, lambda x, y: x * y, 1)

AD.__mul__ = ad_mul

# Define below division, similarly.
# YOUR CODE HERE
def ad_div(self, other):
    return AD._binary_op(self, other, lambda x, y: x / y, 1)

AD.__truediv__ = ad_div

# And finally, define below INTEGER division.
# YOUR CODE HERE
def ad_intdiv(self, other):
    return AD._binary_op(self, other, lambda x, y: x // y, 1)

AD.__floordiv__ = ad_intdiv
```

In [23]:
```
### Tests for multiplication  (10 points for these and the hidden tests)

# Note that the elements that are missing in the other dictionary
# are assumed to be 1.
d = AD(a=2, b=3) * AD(b=4, c=5)
check_equal(d, AD(a=2, b=12, c=5))
```

In [24]:
```
### Tests for division  (10 points for these and the hidden tests)

d = AD(a=8, b=6) / AD(a=2, c=4)
check_equal(d, AD(a=4, b=6, c=0.25))
```

In [25]:
```
### Tests for integer division  (10 points for these and the hidden tests)

d = AD(a=8, b=6) // AD(a=2, b=4, c=4)
check_equal(d, AD(a=4, b=1, c=0))
```

We will let you also implement max_items and min_items properties. The value of these properties should be a *pair*, consisting of the maximum/minimum value, and of the *set* of keys that led to that value. The latter needs to be a set, because there are multiple key, value pairs where the value is maximal or minimal. To avoid wasting your time, we will have you implement only the max_items property. If the AD is empty, the property's value should be the pair `(None, set())`.

In [32]:
```
### Exercise: Implement `max_items`

def ad_max_items(self):
    """Returns a pair consisting of the max value in the AD, and of the
    set of keys that attain that value. This can be done in 7 lines of code."""
    # YOUR CODE HERE
    keys = set()
    max = 0
    if self == {}:
        return None, keys
    else:
        for x in self.keys():
            if self[x] > max:
                max = self[x]
                if keys != set():
                    keys.pop()
                keys.add(x)
            elif self[x] == max:
                keys.add(x)
        return max, keys


# Remember the use of the property decorator.
AD.max_items = property(ad_max_items)
```

In [33]:
```
### Tests for `max_items`  (10 points for these and the hidden tests)

# For empty ADs, it has to return None as value, and the empty set as key set.
```

```
check_equal(AD().max_items, (None, set()))
# Case of one maximum.
check_equal(AD(red=2, green=3, blue=1).max_items, (3, {'green'}))
# Case of multiple maxima.
check_equal(AD(red=2, yellow=3, blue=3, violet=3, pink=1).max_items,
            (3, {'yellow', 'blue', 'violet'}))
```

Comparison and Logical Operators

Comparisons

If `d` is an `AD`, what is the meaning of `d > 2`?

In numpy, when we have an array `a`, and we write `a > 2`, we obtain the array consisting of boolean entries, indicating whether the elements of `a` are greater than two or not:

In [34]:
```
import numpy as np

a = np.array([2, 3, 4, 1, 2])
a > 2
```

Out[34]:
```
array([False,  True,  True, False, False])
```

This suggests adopting a similar semantics for inequalities, defining the behavior of the comparison operators:

In [35]:
```
def ad_lt(self, other):
    return AD._binary_op(self, other, lambda x, y: x < y, 0)

AD.__lt__ = ad_lt
```

In [36]:
```
AD.__lt__ = lambda l, r: AD._binary_op(l, r, lambda x, y: x < y, 0)
AD.__gt__ = lambda l, r: AD._binary_op(l, r, lambda x, y: x > y, 0)
AD.__le__ = lambda l, r: AD._binary_op(l, r, lambda x, y: x <= y, 0)
AD.__ge__ = lambda l, r: AD._binary_op(l, r, lambda x, y: x >= y, 0)
```

Let us see how this works:

In [37]:
```
AD(cat=4, bird=2) > 2
```

Out[37]:
```
{'bird': False, 'cat': True}
```

Once we have a dictionary mapping from values to booleans, we need some way of testing whether all keys, or some keys, map to True. In Python, the `all()` function returns whether all elements in a boolean list are true:

In [38]:
```
print(all([True, False, True, True]))
print(any([True, True, True, True]))
```

```
False
True
```

So we can define the `all` and `any` methods for an AD as follows:

In [39]:
```
def ad_all(self):
    return all(self.values)

AD.all = ad_all
```

In [40]:
```
AD.all = lambda self : all(self.values())
AD.any = lambda self : any(self.values())
```

We also add a method for filtering the elements that satisfy a condition. If the condition is left unspecified, we use the truth-value of the elements themselves.

In [41]:
```
def ad_filter(self, f=None):
    return AD({k: v for k, v in self.items() if (f(v) if f is not None else v)})
AD.filter = ad_filter
```

Let us try this out.

In [42]:
```
d = AD(green=3, blue=2, red=1)
d.filter(lambda x : x > 1)
```

Out[42]:
```
{'blue': 2, 'green': 3}
```

In [44]:
```
d = AD(green=3, blue=2, red=1)
(d > 1).filter()
```

Out[44]:
```
{'blue': True, 'green': True}
```

In [45]:
```
(d > 1).all()
```

Out[45]:
```
False
```

In [46]:
```
(d > 0).all()
```

Out[46]:
```
True
```

Implementing the Sock Drawer In Terms of Arithmetic Dictionaries

In [62]:
```
### Exercise: Implement the sock drawer class using ADs

class ADSockDrawer(object):

    def __init__(self):
        self.drawer = AD()

    def add_sock(self, color):
```

```
        """Adds a sock of the given color to the drawer.
        Do it in one line of code."""
        # YOUR CODE HERE
        if color not in self.drawer:
            self.drawer.update({color: 1})
        else:
            self.drawer[color] += 1

    def get_pair(self, color):
        """Returns False if there is no pair of the given color, and True
        if there is.  In the latter case, it removes the pair from the drawer.
        Do it in 5 lines of code or less."""
        # YOUR CODE HERE
        if self.drawer[color] < 2:
            return False
        else:
            self.drawer[color] -= 2
            return True

    @property
    def available_colors(self):
        """Lists the colors for which we have at least two socks available.
        Do it in 1 line of code."""
        # YOUR CODE HERE
        return {color for color, amount in self.drawer.items() if amount >= 2}
```

In [63]:
```
# Tests for Sock Drawer implementation based on ADs

test_drawer(ADSockDrawer)
```

In [64]:
```
### Hidden tests for Sock Drawer implementation based on ADs  (10 points)
```