1. From class to dataclass

Class `A` has a single attribute `_length` that is not set in the initializer. If you convert this to a data class, you need to use the `field` function and indicate that the `_length` field shouldn't be in the initializer. Secondly, you need to supply the default argument to the `field` function as well so that `_length` is initially 0. This is what that looks like (assuming `_length` is an integer):

```
@dataclass
class A:
    _length: int = field(init=False, default=0)
```

Alternatively, you can add the `__post_init__` dunder method to set the default value, like so:

```
@dataclass
class A:
    _length: int = field(init=False)

    def __post_init__(self) -> None:
        self._length = 0
```

Class `B` has three instance variables including a list of integers. If the list is not provided it's set to the empty list by default. In a dataclass this is achieved using the `default_factory` argument of `field`:

```
@dataclass
class B:
    x: int
    y: str = "hello"
    l: list[int] = field(default_factory=list)
```

Note that with dataclasses you have to make sure that instance variables that don't have a default value are placed above variables with a default value. The reason is that the order in which you define the fields determines the order of the arguments, and arguments without default value can't come after arguments with a default value in Python.

Alright, on to class `C`. Here we have two instance variables, but only the first one is provided and the second one depends on the value of the first one. Here we need to rely on `__post_init__` to assign the value to `b` after `a` has been set, and we also need to make sure that you can't set `b` in the initializer. This is the dataclass that achieves this:

```
@dataclass
class C:
    a: int = 3
```

```
    b: int = field(init=False)

    def __post_init__(self) -> None:
        self.b = self.a + 3
```

As you can see, dataclasses are not always shorter to write than regular classes (the dataclass version of `C` adds an extra line with regards to the regular version). But then again, shortness shouldn't be the only decision factor in whether you should use a dataclass or a regular class, because of the many other benefits that dataclasses offer such as having a nicely printable version of the object and being able to see at a glance what the instance variables are of a dataclass since they're all defined at the top.

Dataclasses also make it easier to use a performance-optimized version of a class using the *slots* mechanism. For example, this is a more performant version of class `B`:

```
@dataclass(slots=True)
class B:
    x: int
    y: str = "hello"
    l: list[int] = field(default_factory=list)
```

2. Let's sell some phones

Here's a possible solution:

```
@dataclass
class Customer:
    name: str
    street: str
    postal_code: str
    city: str
    email :str

@dataclass
class Phone:
    brand: str
    model: str
    price: int
    serial: str

@dataclass
class Plan:
    customer: Customer
    phone: Phone
    start_date: datetime
    duration_months: int
    price: int
    phone_included: bool = False
```

The important thing here is that `Plan` should define the relationship between customers and phones. I've chosen to set the `phone_included` field to `False` by default. Also note that I use integers for representing prices. This is a standard practice in finance (though `Decimal` can also be used). If you use an integer to represent the price, the unit should be the smallest monetary unit for the currency. For example, if the currency is US dollar, the smallest unit is *cent*. So a price of $100.00 is stored as the value 10000 (ten thousand cents).

There are many other ways to define the representation. For example, you could introduce an `Address` class that contains address information and then each customer could have one or more addresses:

```
@dataclass
class Address:
    street: str
    postal_code: str
    city: str

@dataclass
class Customer:
    name: str
    addresses: list[Address] = field(default_factory=list)
    email :str
```

The nice thing about this change is that it now allows customers to have multiple addresses, for example a customer can now have a billing address that's different from the delivery address.