Side Effects:

The problem and its solution using *functional programming*.

Suppose we're implementing a program to handle purchases at a coffee shop. We'll begin with a Python program that uses side effects in its implementation (also called ***impure*** program).

```
class Cafe:
   def buy_coffee(self, cc: CreditCard) -> Coffee:
      cup = Coffee()                  # instantiate a new cup of coffee
      cc.charge(cup.price)            # Charge a credit card with the Coffee's price
                                      # A side effect!!

      return cup                      # Returns the coffee
```

A method call is made on the **`charge`** method of the credit card, resulting in a side effect. Then the cup is passed back to the caller of the method.

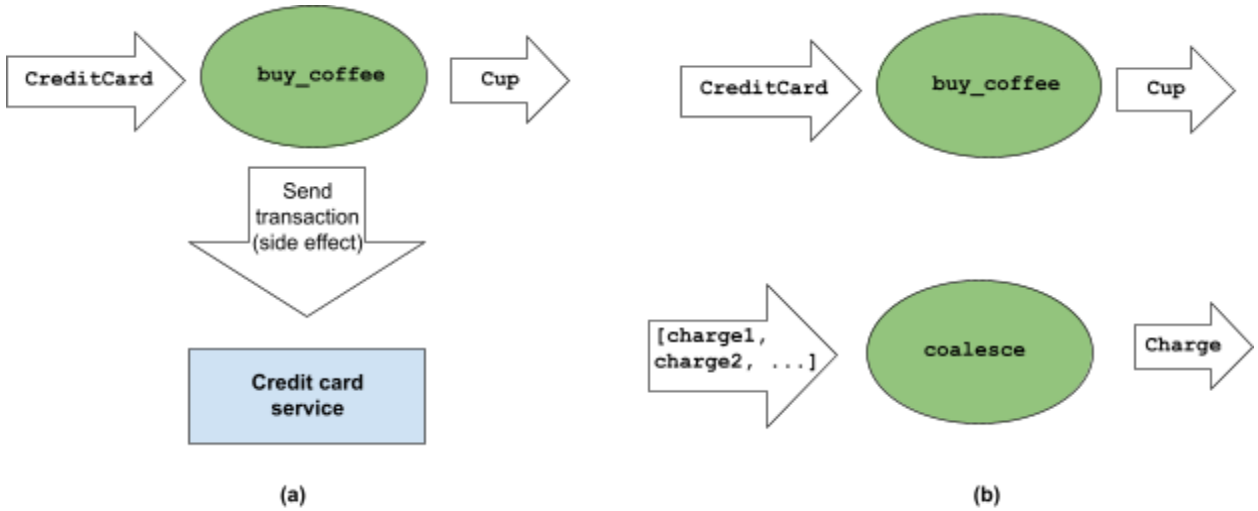
The line **`cc.charge(cup.price)`** is an example of a side effect. Charging a credit card involves some interaction with the outside world. Suppose it requires contacting the credit card provider via some web service, authorizing the transaction, charging the card, and (if successful) persisting a record of the transaction for later reference. In contrast, our function merely returns a **`Coffee`** while these other actions are all happening on the side. Hence the term side effect.
As a result of this side effect, the code is difficult to test. We don't want our tests to actually contact the credit card provider and charge the card! This lack of testability suggests a design change: arguably, **`CreditCard`** shouldn't know how to contact the credit card provider to execute a charge, nor should it know how to persist a record of this charge in our internal systems. We can make the code more modular and testable by letting **`CreditCard`** be agnostic of these concerns and passing a **`Payments`** object into **`buy_coffee`**.

```
class Cafe:
   def buy_coffee(self, cc: CreditCard, p: Payments) -> Coffee:
      cup = Coffee()
      p.charge(cup.price)
      return cup
```

Although side effects still occur when we call `p.charge(cc, cup.price)`, we have at least regained some testability. `Payments` can be an interface, and we can write a mock implementation of this interface suitable for testing. But that isn't ideal either. We're forced to make `Payments` an interface when a concrete class might have been fine otherwise, and any mock implementation will be awkward to use. For example, it might contain some internal state that we'll have to inspect after the call to `buy_coffee`, and our test will have to make sure this state has been appropriately modified (mutated) by the call to charge. We can use a mock framework or similar to handle this detail for us, but this all feels like overkill if we just want to test that `buy_coffee` creates a charge equal to the price of a cup of coffee.
Separate from the concern of testing, there's another problem: it's challenging to reuse `buy_coffee`. Suppose a customer, Alice, would like to order 12 cups of coffee. Ideally, we could just reuse `buy_coffee` for this, perhaps calling it 12 times in a loop. But as it is currently implemented, that will involve contacting the payment provider 12 times and authorizing 12 separate charges to Alice's credit card! That adds more processing fees and isn't good for Alice or the coffee shop.

What can we do about this? We could write a whole new function, `buy_coffees`, with particular logic for batching the charges. Here, that might not be a big deal since the logic of `buy_coffee` is so simple; but in other cases, the logic we need to duplicate may be nontrivial, and we should mourn the loss of code reuse and composition!

The functional solution is to eliminate side effects and have `buy_coffee` return the charge as a value in addition to returning `Coffee`, as shown in figure 1.1. The concerns of processing the charge by sending it to the credit card provider, persisting a record of it, and so on, will be handled elsewhere.



A call to `buy_coffee`, **(a)** with and **(b)** without a side effect.

```
class Cafe:
   def buy_coffee(self, cc: CreditCard)-> tuple(Coffee,Charge):
      cup = Coffee()
      return (cup, Charge(cc, cup.price))
```

We've separated the concern of *creating* a charge from the *processing* or *interpretation* of that charge. The **`buy_coffee`** function now returns a **`Charge`** as a value along with **`Coffee`**. We'll see shortly how this lets us reuse it more easily to purchase multiple coffees with a single transaction. But what is **`Charge`**? It's a data type we just invented, containing a **`CreditCard`** and an amount, equipped with a handy combine function for combining charges with the same **`CreditCard`**.

```
class Charge:
   def __init__(self, cc: CreditCard, amount: Float):    #1
      self.cc = cc
      self.amount = amount

   def combine(self, other: Charge):    #2
      if self.cc == other.cc:    #3
         return Charge(self.cc, self.amount + other.amount)  #4
      else:
         raise Exception("Cannot combine charges to different cards")
```

1) Data class declaration with a constructor and immutable fields
2) Combines charges for the same credit card
3) Ensures that it's the same card; otherwise, throws an exception
4) Returns a new **`Charge`**, combining the amount of this charge and the other

This data type is responsible for holding the values for a **`CreditCard`** and an amount of Float. A handy method is also exposed that allows this **`Charge`** to be combined with another **`Charge`** instance. An exception will be thrown when an attempt is made to combine two charges with a different credit card.
Note: The throwing of an exception is not ideal, and we'll discuss more functional approaches to handling error conditions.

Now let's look at **`buy_coffees`** to implement the purchase of **`n`** cups of coffee. Unlike before, this can now be implemented in terms of **`buy_coffee`**, as we had hoped.

```
class Cafe:
   def buy_coffee(self, cc: CreditCard) -> Tuple[Coffee, Charge] :
      pass
   def buy_coffees(self, cc: CreditCard, n: int) -> Tuple[List[Coffee], Charge]:
      purchases: List[Tuple[Coffee, Charge]] = [buy_coffee(cc) for i in
range(len(n))]
      coffees, charges = zip(*purchases)
      return (coffees, reduce(lambda x, y: x + y, numbers))
```

The example takes two parameters: a **`CreditCard`** and the **`Int`** number of coffees to be purchased. After the **`Coffees`** have been successfully purchased, they are placed into a **`List`** data type. The list is initialized using (list comprehension syntax):
`[buy_coffee(cc) for i in range(len(n))]`
where **`n`** describes the number of coffees and **`buy_coffee(cc)`** is a function that initializes each element of the list.

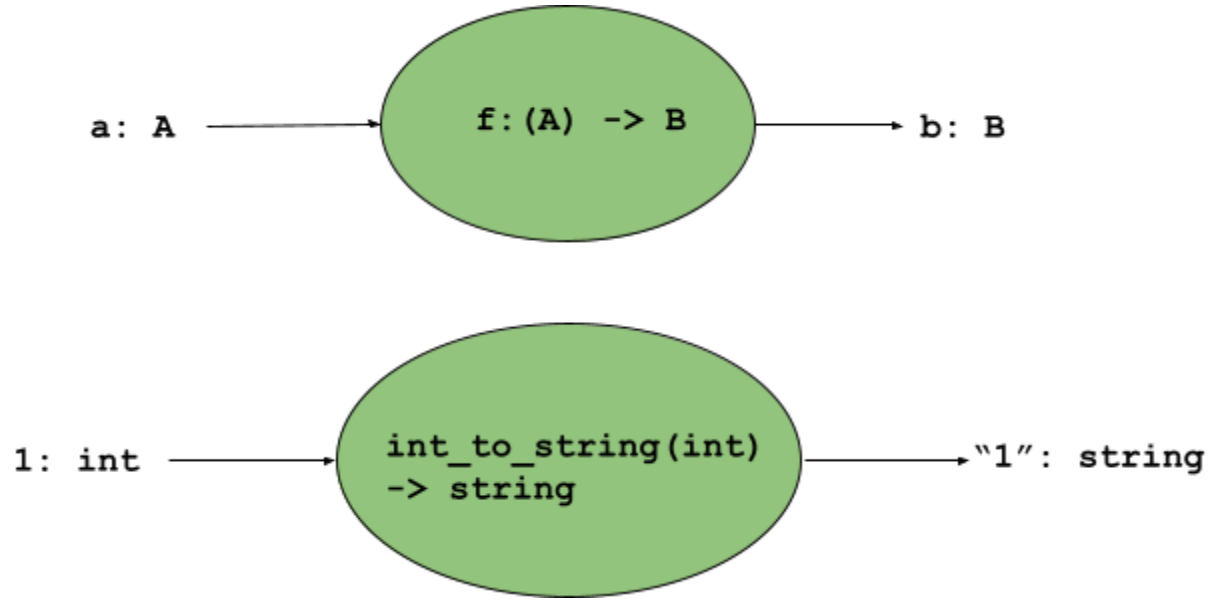
The **`zip(*purchases)`** is then used to *destructure* the list of tuples into two separate lists, each representing one side of the pair. *Destructuring* is the process of extracting values from a complex data type. We are now left with the coffees list being a **`List[Coffee]`** and charges being a **`List[Charge]`**. The final step involves reconstructing the data into the required output. This is done by constructing a Pair of **`List[Coffee]`** mapped to the combined **`Charges`** for all the **`Coffees`** in the list. reduce is an example of a higher-order function, which we will introduce appropriately later.

Overall, this solution is a marked improvement—we're now able to reuse **`buy_coffee`** directly to define the **`buy_coffees`** function. Both functions are trivially testable without defining complicated mock implementations of a **`Payments`** interface! In fact, **`Cafe`** is now wholly ignorant of how the **`Charge`** values will be processed. We can still have the **`Payments`** class for actually processing charges, of course, but **`Cafe`** doesn't need to know about it. Making **`Charge`** into a first-class value has other benefits we might not have anticipated: we can more easily assemble business logic for working with these charges. For instance, Alice may bring her laptop to the coffee shop and work there for a few hours, making occasional purchases. It might be nice if the coffee shop could combine Alice's purchases into a single charge, again saving on credit card processing fees.

Exactly what is a (pure) function?

Functional Programming means programming with pure functions, and **a pure function lacks side effects**. In our discussion of the coffee shop example, we worked using an informal notion of side effects and purity. Here we'll formalize this notion to pinpoint more precisely what it means to program functionally. This will also give us additional insight into one benefit of FP: pure functions are easier to reason about.

A function **`f`** with input type **`A`** and output type **`B`** is a computation that relates every value **`a`** of type **`A`** to exactly one value **`b`** of type **`B`** such that b is determined solely by the value of **`a`**. Any changing state of an internal or external process is irrelevant to compute the result **`f(a)`**. For example, a function **`int_to_string`** having type **`(int) -> string`** will take every integer to a corresponding string. Furthermore, if it really is a function, it will do nothing else.



In other words, if a function has no observable effect on the execution of the program other than to compute a result given its inputs, we say that it has no side effects.