Classes

The idea of capturing state in a function using a closure leads to the concept of **classes**, and **instances** of these classes.

In Python, a class is simply defined as:

```
class MyClass:
    pass
```

which is equivalent to saying

```
class MyClass(object):
    pass
```

In python, everything is an object: functions, integers, class instances, etc, and they all "derive" from the object **object**. This language will be clearer later.

Mutating State

```
class BankAccount:

    def __init__(self, balance):
        self.balance = balance

    def withdraw(self, amount):
        self.balance = self.balance - amount
```

We make an *instance* of a *class* `BankAccount` called `myaccount` with a balance of 100 and withdraw 20 from it. Both the "function" `withdraw` and the variable representing the `balance` are called as if they belong to the instance `myaccount`.

```
myaccount = BankAccount(100)
print(myaccount.balance) # 100
myaccount.withdraw(20)
print(myaccount.balance) # 80
```

- `__init__`: a constructor for the class. This is the function called when we say `BankAccount(100)`.
- Why does `__init__(self, balance)` have 2 arguments then? This is because it is a very special kind of function called a **method**, in which the first argument is the *instance* of the class. By convention, it is always called `self` in python. Thus we will use `BankAccount(balance)` to call the *constructor method*.
- `withdraw(self, amount)`: another *method*. Once again `self` is the existing account object. You can think of your program's `myaccount` instance `withdraw`-ing the `amount` and thus write it `myaccount.withdraw(amount)`, the implicit `self` having been moved to the left of the dot.
- `myaccount.balance`: this is an *instance variable*, some data 'belonging" to the instance, just as the previous method did. Thus we'll use `self.balance` inside the methods to denote it.

Class Variables and Class Methods

How do we share variables and functionality across all instances of a class?

```
class BankAccount:
    max_balance = 1000

    @classmethod
    def make_account(cls, balance):
        if balance <= cls.max_balance:
            return cls(balance)
        else:
            raise ValueError(f"{balance} too large")

    def __init__(self, balance):
        self.balance = balance

    def withdraw(self, amount):
        self.balance = self.balance - amount

ba = BankAccount.make_account(100)
ba.balance # 100
ba.withdraw(20)
ba.balance # 80

bb = BankAccount.make_account(10000) # ValueError: 10000 too large
```

- `max_balance` is a **class variable** since this max value needs to be shared by all accounts. Note it is not preceded with a `self`.
- The **classmethod** `make_account`, announced thus by *decorating* it with `@classmethod`, takes the class, NOT the instance, as its "implicit" (as in not written) first argument, moving the class over to the left side of the dot.
- It calls the constructor as `cls(balance)` if the class variable `cls.max_balance` is reasonable.

Inheritance

```
class BankAccount: # BankAccount(object)
    max_balance = 1000

    @classmethod
    def make_account(cls, balance):
        if balance <= cls.max_balance:
            return cls(balance)
        else:
            raise ValueError(f"{balance} too large")
    def print_balance(self):
        print("Balance", self.balance)
    def __init__(self, balance):
        self.balance = balance

    def withdraw(self, amount):
        leftover = self.balance - amount
        if leftover >= 0:
            self.balance = leftover
        else:
            raise ValueError("Withdrawal not Allowed!")

ba = BankAccount.make_account(100)
ba.print_balance() # 100
ba.withdraw(20)
ba.print_balance() # 80
ba.withdraw(100) # gives: `ValueError: Withdrawal not Allowed!`
```

All classes implicitly inherit from `object`

```
class VIPBankAccount(BankAccount):

    def __init__(self, balance):
        super().__init__(balance)
        # super() equivalent to: super(VIPBankAccount, self)
        self.balance += 10 # 10 free dollars

    def withdraw(self, amount):
        leftover = self.balance - amount
        if leftover >= 0:
            self.balance = leftover
        else:
            print("You'vegone negative")
            self.balance = leftover

ba2 = VIPBankAccount.make_account(100)
ba2.print_balance() # 100
ba2.withdraw(20)
ba2.print_balance() # 80
ba2.withdraw(100) # prints `You've gone negative`.
```

The class `OverdrawAllowingBankAccount` *inherits* from `BankAccount`. We utilize parent or "base" classes' constructor and `print_balance`, and redefine `withdraw` in the child class.