Because I’ve struggled with a few concepts throughout my experience with code, I decided to jump into the waters of Python Interfaces. Unfortunately, with my research, I’ve found some bad news: Python does interfaces oddly. That’s okay though because the grandiose view of interfaces is much more clear to me now because of the perspective Python puts on it.

For those that don’t know (I can finally say that! YESSS!), an interface is a programming construct the defines a template (sometimes referred to as a contract) that implementing classes will be required to follow at minimum. For instance, if we have an interface for food named IFood with a method *prepare()* defined, then we can make our classes Burger, Pizza, Taco, and IceCream implement IFood and expound on it.

IFood.prepare() would be considered an abstract method; that is, one that is defined, but doesn’t do anything yet. When we implement IFood in, say, Burger, Burger is required to define the method’s specific functionality.

For further example, since NMC has historically been a .NET Framework institution, you would see something like   
  
interface IFood{

bool prepare();

}

public class Burger : IFood{

bool prepare(){

//TODO: Assemble Burger

}

}

public class Pizza : IFood{

bool prepare(){

//TODO: Cover in toppings!

}

}

//TODO: ADD MORE!

Python, though, is not statically typed, meaning we don’t need to define our variable types, nor our returns, so interfaces become a bit murky. The term used to refer to how Python does this is “Duck Typing” which state that “If it looks like a duck, quacks like a duck, and walks like a duck, then it’s probably a duck.” Leaving the following programmers to have to deduce where methods are created based on later (joyous… day…).

The same concept define in C# previously would be written in this way for Python:

# import our relevant module

from abc import ABCMeta, abstractmethod

# this class will “act” as our interface (Python is weird remember)

class IFood(metaclass=ABCMeta):

@abstractmethod # abstract methods will be defined more verbosely later

def prepare(self):

pass # pass bypasses the need for functionality but satisfies the method declaration and indentation requirements

# much like how you would subclass (pssssttt… that’s all Python interfaces are)  
# we’re forcing Burger to be a subclass of IFood

class Burger(IFood):

def prepare(self): # implement the method defined in our blueprint

print("ADD INGREDIENTS") # Do the thing!

# now prove that our subclass/concrete class functions as an implementer of the abstract superclass

burger = Burger()

burger.prepare()

This allows us to force code standards and iterative updates to like-designed objects such as vehicles, foods, seating furniture, etc. You can extrapolate this out further by creating interfaces that implement interfaces prior

Think like the animal classification system (a classic reference for textbooks)

* IKingdom
  + IPhylum
    - IClass
      * IOrder
        + IFamily

IGenus

ISpecies

Each interface is more specific than the last, allowing each of your subsequent objects to be more granularly defined. It’s worth noting, however, that in designing programs you’ll want to make sure you’re not tightly locking down your structure, but rather allowing permutations of the program to be developed over time.

References:

<https://realpython.com/python-interface/>

<https://www.kite.com/python/answers/how-to-implement-an-interface-in-python>

<https://stackoverflow.com/questions/56657059/why-i-cant-import-abc-but-abcmeta-is-correctly-imported>

<https://stackoverflow.com/questions/6802573/interfaces-whats-the-point>