Classes:

This is in a sense, when we begin creating our own "Datatypes".

For instance, a string has some features, like length. A human has some features like height, weight, How annoying they are etc. These are called Attributes. They are characteristics of the object in question.

"Object" refers to one entity belonging to the group called the "Class".

John is an object belonging to class Human.

Duke is an object belonging to class Dog.

"Dipshit" is an object belonging to class string, and so on.

Creating classes, and maintaining attributes is a very convenient way to handle data.

```
class Human:
      def saysomething(self):
            print("Something")
```

the functions a human can DO are called "methods", and they can return values, do stuff etc just like ordinary functions. The first argument of a method is always the "self", denoting that this is a procedure any human can follow, but the person who is going to do it right now, is John.

```
X=Human()
X.saysomething()
```

That's how a method is called. You do objectname.methodname(non self arguments) (Arguments are the numbers or variables you give to a function before you ask it do something. Here, there are no arguments.)

To define the value of an attribute, we do:

```
X.name="John"
print(X.name)
```

If you want to read the value of an attribute, there are NO brackets.

Defining attributes one by one like this is kinda stupid, so the moment you create the object, you also assign the attributes in a compact way using something called the constructor. This is given the name initiator in python, and is denoted __init__

With double underscores on both sides for bonus fanciness. The functions with such double underscores have internal significance and are known as dunder functions.

```
class Human:
```

```
    def __init__(self,inputname,inputheight):
        self.name=inputname
        self.height=inputheight

    def saysomething(self):
        print("Something")
```

The self.name variable denotes that the object in question has acquired an attribute called name, and its value is given by the inputted value inputname. Conventionally, the input arguments, as well as the attributes, are given the same name, which looks a bit weird at first, but you very quickly get used to it.

Now, we can do:

```
X=Human("John",185)
```

We can still add new attributes directly to object X, by the old way, or even modify existing attributes.

Now that I'm happy with the basic things, let's create subclasses. This is again, a tool for convenience. How so? Somebody else may have created class Human, let's say Tony stark made this class, and published it.

Say, I now want to create an object for Spiderman, or Doctor Strange. They *do* belong to class Human, but I want to give them some special attributes, and methods. Since they do belong to class Human, they should a) have all the same attributes of a Human, and b) do anything a human can do, that is, inherit the *methods* defined for a human.

I don't want to have to copy paste code from tony stark just to achieve this. Besides, if I want to have human objects too, that's large volumes of repeated code, apart from it failing to capture the fundamental idea that Avengers are a subclass of Humans.

```
class Avenger(Human):
    pass
```

This presently means that Avenger, is a subclass of class Human, and everything that works for Human, works here too. In fact, one of the defining features of a *sub*class is that it is in a way, a *superset*. You can replace every human object with an avenger, and no code will break, because an Avenger can do anything a human can do.

This statement however is more apt for a language like Java, rather than Python, but we will get into why that is so when we deal with Java.

```
X=Avenger("Bruce",243)
```

Here, the Avenger class inherited even the init function. If we redefine it, then the new definition given inside class Avenger *overrides* the parent class' initiator.

```
class Avenger(Human):

      def __init__(self,name,height,catchphrase):
            self.name=name
            self.height=height
            self.catchphrase= catchphrase

      def speak(self):
            print(catchphrase)

TS=Avenger("Tony",185,"I am Iron man.")
TS.saysomething()
TS.speak()
```

Okay, looks good. However, we are repeating init code. Can't we streamline that? Yes, we can.

```
class Avenger(Human):

      def __init__(self,name,height,catchphrase):
            super().__init__(name,height)
            self.catchphrase= catchphrase

      def speak(self):
            print(catchphrase)
```

The super function allows you to build on top of a method of the parent class very elegantly.

I'm now going to stop the superhero theme and jump into an example I found in this cool talk.

```
import math
class Circle:

      version = '0.1'

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

      def area(self):
            return math.pi * self.radius ** 2

      def perimeter(self):
            return 2* math.pi * self.radius
```

Note 1: The version number is a string. Why? Is version 0.2 + 0.1 = 0.3? No, it's version 0.30000000000000004. To be honest, nobody should be doing addition with that number. That thing is just a string. Keep it that way.

Note 2: we called math.pi for pi, instead of defining it ourselves. Why? a) it's less effort b) pi isn't a constant, it's a variable that never changes. When you run the code on a 64 bit machine it will use the 64 bit Pi. If you run it on a 32 bit machine, math.pi will adapt. It's maximising code reuse and compatibility.

Alright, we get ourselves a customer, and we send off this code.

```
import circle

bounding_box_diagonal=25.6
c = Circle(bbd_to_radius(bounding_box_diagonal)
```

And they keep doing this all over their code. They have this converter function sticking around every single object initiation they have in all their code. Is that a pain? Yes.

Did you know that the datetime module let's you init in more than one way?

```
x= datetime(1947, 8, 15)
y= datetime.now()
z= datetime.fromtimestamp(555964200)
```

Oh, and by the way, I just spent 10 minutes trying to find out why the x.timestamp() method threw an OS error. Apparently, the timestamp is the number of seconds from January 1st 1970 00:00:00 UTC. 1947 doesn't *have* a timestamp. Oh, also, a lot of computers use only a 32 bit timestamp, so once the number of seconds gets too big, that is on, 19 January 2038, the world will face great catastrophe. Just kidding; it's no biggie.

Back to the topic at hand, Multiple constructors: if your customers need it, you better make it. How?

```
import math
class Circle:

	version = '0.2'

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

	@classmethod
	def frombbd(cls,bbd):
		return Circle(bbd/(2*math.sqrt(2)))

	def area(self):
```

```
            return math.pi * self.radius ** 2

      def perimeter(self):
            return 2* math.pi * self.radius
```

That's a *class method*. You don't need to make an object to call that method. That now returns an object we need so we can live happily ever after.

Except, I was dumb enough to return Circle(bleh), when I took the class, "cls" as an input to the class method. That's the alternative to self for a class, by the way. Why is this dumb? Because the people who use your code will subclass your code. They'll make their own constructors and such, but they will now run into trouble if they use this inherited class method.

They'll expect a gold plate, a donut, or a rubber tube with a duck on it, but they'll get a plain ol' circle. Disastrous.

That's why we should be returning cls(bleh).

This would be fine if the subclass init only made derived declarations, like inner_donut_ring=0.5*radius, but what if the subclass has an init function which requires more arguments? Then, this cls(bleh) throws an error, as it rightfully should, instead of sneakily making a circle and waiting around until something breaks.

Alright, now we ship this code to a rubber tyre company.

```
tyre = Circle(5)
print("Cold area of the tyre is",tyre.area())
tyre.radius *= 1.5
print("Warm area of the tyre is",tyre.area())

class Tyre(Circle):
      def perimeter(self):
            return Circle.perimeter(self) * 1.5
```

Okay, what have they done? Looks like rubber expands on heating, so they went and changed the radius parameter. Nothing's wrong with that, right? Well, most other languages, like Java or C++ actually will be aghast at even the suggestion of that.

Why? Variables used inside a class must, according to them, remain an internal construct, inaccessible from outside the class, that is, a "private" variable. If these parameters are changed freely, people might break code by changing parameters, or put code in an impossible to logically reach state.

Python offers complete freedom. It asks the programmer to be disciplined enough to not be dumb. Is that a good idea? Well, that's what python thinks anyway.

Tomorrow, if version 0.3 of the Circle package adopts r as the variable name to store the radius, all of the tyre companies' code will break. They must live with that, because that's how bad this code is. Let me illustrate the issue further.

This tyre company sold tyres to an airplane company, and because the area() method used self.radius, and a lot of other airplane calculations have area based off of the tyre's perimeter, the inconsistency between the area found in the two ways lead to an airplane crash. Now the government puts out a mandate that area shall not be computed with radius, but instead from the perimeter.

Quite an elaborate backstory, but this is how it turns out:

```
import math
class Circle:

	version = '0.2'

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

	@classmethod
	def frombbd(cls,bbd):
		return Circle(bbd/(2*math.sqrt(2)))

	def area(self):
		p = self.perimeter()
		r = p / (2*math.pi)
		return math.pi * r ** 2

	def perimeter(self):
		return 2* math.pi * self.radius
```

Happily ever after? No. The Tyre company overrode our perimeter function in their subclass declaration. Now the area will be screwed over.

Why did this happen? Because the area method depends on the perimeter method, which was an internal method, which we opened up to be editable and overridden by the subclass. Ideally, the perimeter method must have been a "protected" method. But python lacks any sense of privacy. So, how does python propose to fix this mess?

Well, forget about the python god coming to help you, before that, you can do something yourself. You can just declare a class variable called _perimeter = perimeter at the end of the class declaration, and call this _perimeter function internally in the area method. Now, the Tyre company can go and override the method all they want. You have a spare copy.

Well, if that's a good idea for you, it's a good idea for them. The Tyre company wants to have a protected copy of their perimeter function against their subclasses, so they define _perimeter for safety, once again, utterly destroying the integrity of your code.

Okay, so now what. Just call it Circle_perimeter, and the tyre guys can call their protected variable Tyre_perimeter. Everyone's happy. This works perfect, so python has automated this unique variable naming for you. That's accomplished with double underscores before the start; _ _perimeter can now be used by everyone to make their protected copies.

That's actually not a bad solution to not having privacy in methods. Privacy in variables is still something you might want. In other languages, people design individual functions to get and to set parameters internally. If you wanted to set the radius, you don't do c.radius=5, but instead c.setradius(5). You don't call c.radius, you use c.getradius().

These are called getters and setters and they form the interface into the class, while allowing the change of parameters in a legal and acceptable fashion. Even if the underlying implementations in a class are modified, the API provided by the class methods will stand strong.

Just to illustrate this, consider a new government regulation: You're forbidden from storing the radius of a circle. You're only allowed to store the diameter. What does this break? Well, pretty much everything. Every function needs to be rewritten, but since the API is the same, the clients won't notice anything. Except, c.radius doesn't exist anymore. And they cannot set it directly, because it's a derived quantity.

There's actually a way to avoid having to rewrite every function. What if c.radius just called a getradius function, like Java or C++ would, and inside the getter function, we return self.diameter/2. Nothing would break. Similarly, we need a setradius function that sets self.diameter = 2*r_input.

This is the point where python admits getters and setters are a sensible idea.

```
import math
class Circle:

	version = '0.3'

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

	@classmethod
	def frombbd(cls,bbd):
		return Circle(bbd/(2*math.sqrt(2)))

	@property # Converts dotted access to method call
	def radius(self):
		return self.diameter/2
```

```
    @radius.setter
    def radius(self,radius):
        self.diameter = radius*2

    def area(self):
        p = self.perimeter()
        r = p / (2*math.pi)
        return math.pi * r ** 2

    def perimeter(self):
        return 2* math.pi * self.radius
```

Pretty nifty that python has an inbuilt decorator for the job. This lets you have the freedom and cleanliness of dotted access and never having to design unnecessary getters and setters, while choosing to make them for variables that really do need it. You won't even notice they exist if you don't look inside the class.

Aside from the 3 decorators we've seen, there's one more: @staticmethod. Often, people who work with circles, need to convert degrees to radians all the time. What do they do? They go searching for helper functions from some other place. Wouldn't it be nice if they could just call circle.deg_to_rad()? All the code that's relevant will be easy to access and discover.

This kind of a method that has nothing to do with the "self" object, is called a static method. It's really a function, but it has been made a method for the sake of accessibility.

Summary:

1. Instance variables have information unique to an instance.
2. Class variables are shared among all instances
3. Regular methods use self to operate on instance data.
4. Class methods use cls to implement alt constructors that also work on subclasses.
5. Static methods do not use self or cls. They improve discoverability of relevant code.
6. A property decorator converts a method into a getter function. If and only if a property getter has been defined in this manner, a setter function can be defined with the attribute.setter decorator.