DATACLASSES

The Python Bakery

Primitives Types

- Boolean
- Integer
- Float
- String
- None

We have learned about 5 primitive types: Booleans, Integers, Floats, Strings, and the special None.
We can use these to represent many kinds of data, but they have a limitation since they are so simple and cannot be combined with each other.
To increase our capabilities, we will learn about a new category of types, which is Composite types.
Composite types are named because they are composed of other types, unlike primitive types.
The first kind of composite types we will learn about are Dataclasses.

Dataclasses

- Dataclass: A data structure that organizes variables together in one place.

Dataclasses are a data structure that allow us to combine other types together to make a new type.
You should not think of Dataclass as a type itself; instead, it is a way of creating new types.
Or put another way, dataclasses are a kind of type, without being a type themselves.

A First Example

```
# Required import
from dataclasses import dataclass

# Definition
@dataclass
class Box:
    width: int
    length: int

# Instance creation
my_box = Box(5, 10)
# Using the instance
print(my_box)
```

Let's take a look at an example Dataclass.
First, we must import a decorator from the dataclasses module.
Then we create a definition for the dataclass by writing the header and body of the dataclass.
The header requires the decorator, the class keyword, the name of the new dataclass, and a colon.
The body requires the we specify the fields, each on their own line.
Once defined, we can create instances of the dataclass using the constructor function, which has the same name as the defined dataclass.
Storing the result in a variable allows us to use the instance later, such as shown here where we print the my_box variable.
Each part of this example needs to be broken down further.

@dataclass Decorator

```
from dataclasses import dataclass

@dataclass
class Dog:
  pass
```

Perhaps the most confusing part of dataclasses is the decorator.
The decorator is the @dataclass annotation.
The "at" symbol that we see in front of the word "dataclass" is called a decorator.
We are not going to explain what this does in-depth or why we need it, but we definitely do need it.
In short, a decorator adds extra functionality on top of classes to make them dataclasses, which are far more powerful and convenient.
There's a lot that can go wrong here, when using a decorator.
You must import the dataclass decorator from the dataclasses module, without including the @ symbol.
However, when you put the decorator above the class definition, you absolutely must include the @ symbol.
Messing up either part will prevent Python from understanding that you want to create a dataclass.

Dataclass Fields

```
from dataclasses import dataclass

@dataclass
class Dog:
    name: str
    age: int
    is_fuzzy: bool
```

The fields of a dataclass are the most useful part, since they allow us to bundle up related data into a single place.
Each field is like a variable inside of the dataclass.
The rules for naming fields are the same as the rules for naming variables.
First, the name must have only letters, digits, and underscores.
Second, the name must only begin with letters or underscores.
The rules for specifying the type of fields is similar to the rules for specifying the type of function parameters, except instead of commas each field is instead on its own line.

The Constructor Function

```
from dataclasses import dataclass

@dataclass
class Dog:
    name: str
    age: int
    is_fuzzy: bool

ada = Dog("Ada Bart", 4, True)
```

Once a dataclass is defined, we can create any number of instances that represent concrete versions of the original dataclass.
Instances of a dataclass are created by calling the constructor function.
The constructor is a special function created by the dataclass definition, and given the same name.
The similarity between defining a function and defining a dataclass is very strong.
The similarity between calling a function and instantiating a dataclass is even stronger.
Each field of the dataclasses definition corresponds to an argument for the constructor.
The order and type of each field must match the order and type of each argument.

Class vs Instance

- Class: The pattern that is used as a base for instances
- Instance: A specific, concrete example of a class

A very confusing concept is what a class represents compared to an instance of the class.
A similar concept is the difference between a stencil and a drawing.
Another similar concept is the difference between a recipe and a cake.
In all these cases, one of these is an idea, something totally theoretically.
The other is a concrete thing.
As an example, let us consider the Box example from before.
When you only look at the definition of the Box, it is impossible to say what color a Box is.
You can generally say that the color of a Box will be a string value.
But there is no specific string value associated with the idea of a Box.
Instead, you would need to create an instance of a Box in order to talk meaningfully about the color of that box.
The concept of a Box does not have a color.
You can create an instance using the constructor function that is "red" or "blue", but the original Box object itself does not inherently have a color.

Accessing Fields

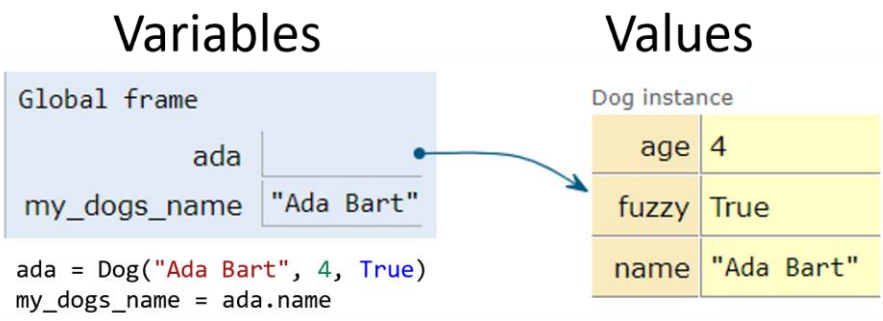
```
from dataclasses import dataclass

@dataclass
class Dog:
    name: str
    age: int
    is_fuzzy: bool

ada = Dog("Ada Bart", 4, True)
print("Ada's name is", ada.name)
```

One of the most critical part of using an instance is to access its fields.
The period and the name of the field are used to access those fields.
Generally, you can think of a field as like a variable that lives inside of an instance.
In the code shown here, we access the name field of the Dog instance stored in ada.
A critical thing to understand, however, is that you can only access the fields of an instance, not a dataclass.
In other words, you cannot try to access the fields of the Dog dataclass, only the ada instance.
If you try changing the code, you will see that you get an error when you try to access Dog.name instead of ada.name.
It's very important to understand the idea that the Dog is an abstract idea, while the ada instance is a real concrete bit of data.

Tracing Dataclasses




```
ada = Dog("Ada Bart", 4, True)
my_dogs_name = ada.name
```

In the Variables/Values diagram shown here, you can see how Dataclasses are traced a little differently compared to regular primitive types of data.
When the constructor is called, a new instance of a dataclass is created.
That instance is a special kind of value with multiple fields, which can be seen on the right-hand side of this diagram.
When code accesses one of those fields, like on the second line shown, the arrow is followed and the relevant field's value is chosen.
That value is a simple primitive string, and so can be stored directly in a variable.
These arrows actually have a lot more significance than we want to talk about right now, but the important thing to recognize is that ada variable points to a much more complex value than the my_dogs_name variable.