5.2 Defining Our Own Data Types, Part 1

Up to this point, all the data we've worked with in Python have been stored in objects that are instances of the built-in types that come with Python, like `int`s and `list`s. Python's built-in data types are powerful, but are not always the most intuitive way to store data. For example, we saw in 5.1 Tabular Data that we could use a list of lists to represent tabular data. One of the downsides of this approach is that when working with this data, the onus is on us to remember which list element corresponds to which component of the data.

```
>>> import datetime
>>> row = [1657, 'ET', 80, datetime.date(2011, 1, 1)]
>>> row[0]  # The id
1657
>>> row[1]  # The name of the civic centre
'ET'
>>> row[2]  # The number of marriage licenses issued
80
>>> row[3]  # The time period
datetime.date(2011, 1, 1)
```

You can imagine how error prone this might be. A simple "off by one" error for an index might retrieve a completely different data type. It also makes our code difficult to read; the reader must know what each index of the list represents. And, as more experienced programmers will tell you, readable code is crucial.[1]

So a row in our marriage license data set is made up of four data elements. It would be nice if, instead of indices, we could use names that were reflective of each piece of data. Certainly, we could use a dictionary (instead of a list), with string keys naming each piece of data. But there is a more robust option we'll learn about in this section: creating our *own* data types.

[1] "Any fool can write code that a computer can understand. Good programmers write code that humans can understand." – Martin Fowler

Defining a data class

You might remember from 2.1 Python's Built-In Functions that in Python, another term for "data type" is **class**.[2] The built-in data types we've studied so far illustrate how rich and complex data types can be in Python. So in learning to create our own data types, we will first learn about the simplest kind of data type: a **data class**, which is a kind of class whose purpose is to bundle individual pieces of data into a single Python object.

For example, suppose we want to represent a "person" consisting of a given name, family name, age, and home address. We already know how to represent each individual piece of data: the given name, family name, and address could be strings, and the age could be a natural number. To bundle these values together, we could use a list or other built-in collection data type, but that approach would run into the issues we discussed above.

So instead, we define our own data class to create a new data type consisting of these four values. Here is the way to create a data class in Python:

```
from dataclasses import dataclass


@dataclass
class Person:
    """A custom data type that represents data for a perso
    """
    given_name: str
    family_name: str
    age: int
    address: str
```

[2] This is why `type(3)` evaluates to `<class 'int'>` in Python.

Let's unpack this definition.

1. `from dataclasses import dataclass` is a Python import-from statement that lets us use `dataclass` below.
2. `@dataclass` is a Python *decorator*. We've already seen decorators for function definitions when using the `hypothesis` library for property-based testing. A decorator for a class definition works in the same way, acting as a modifier for our definition. In this case, `@dataclass` tells Python that the data type we're defining is a data class.
3. `class Person:` is the syntax for the start of a *class definition*.[3] The name of the data class is `Person`.

   The rest of the code for the class is indented to put it inside of the class body.
4. The next line, `"""A custom data type...` is a docstring that describes the purpose of the data class.
5. Each remaining line (starting with `given_name: str`) defines a piece of data associated with the data class; each piece of data is called an **instance attribute** (often shortened to just **attribute**) of the data class.

   For each instance attribute, we write a name and a type annotation.[4]

[3] This is similar to function definitions, except we use the `class` keyword instead of `def`.

[4] This is similar to defining function parameter names and types, though of course the purposes are different here.

Data class definition syntax

In general, a data class definition in Python has the following syntax:

```
@dataclass
class <ClassName>:
    """Description of the data class.
    """
    <attribute1>: <type1>
    <attribute2>: <type2>
    ...
```

Using data classes

Now that we've seen how to define a data class, we now are ready to actually put it to use. For built-in Python data types, we know how to create values of those types: type in literals like `3` or `['hi', 'bye']`. But with our `Person` data class, what is the corresponding literal we can write?

The answer is there isn't—the possible literals of the Python programming language are fixed, and can't be changed even after defining a new data type. But all is not lost! By defining a `Person` data class, we have gained the ability to *call the data class like a function* to create values whose type is `Person`.[5] Here is an example of creating a `Person` value, passing in as arguments the values for each instance attribute:

```
>>> david = Person('David', 'Liu', 100, '40 St. George Str
```

Pretty cool! That line of code creates a new `Person` value whose given name is `'David'`, family name is `'Liu'`, age is `100`, and address is `'40 St. George Street'`, and stores the value in the variable `david`. The *type* of this new value is, as we'd expect, `Person`:

```
>>> type(david)
<class Person>
```

[5] We've actually seen this before: we create `range` values by calling `range` like a function, e.g. `range(1, 10)`, and just in the last section we created `datetime.date` objects like `datetime.date(2011, 1, 1)`.

One new piece of terminology: we say that the value `david` refers to is an **instance** of the `Person` data class. In other words, the phrases "`david` has type `Person`" and "`david` is an instance of `Person`" mean the same thing. This explains why we refer to the bundled data in a data class as *instance* attributes, since they are pieces of data that are associated with a particular value of that type.

Accessing instance attributes

If we evaluate the `Person` value, we see the different pieces of data that have been bundled together:

```
>>> david
Person(given_name='David', family_name='Liu', age=100, address='40 St. George Street')
```

But from a `Person` value, how do we extract the individual pieces of data we bundled together? If we were using lists, we'd simply do list indexing: `david[0]`, `david[1]`, etc. The syntax for Python classes improves this because we can use the names of the instance attributes together with **dot notation** to access these values:

```
>>> david.given_name
'David'
>>> david.family_name
'Liu'
>>> david.age
100
>>> david.address
'40 St. George Street'
```

This is much more readable than list indexing, and this is one of the major advantages of using data classes over lists to represent custom data in Python.

Tip: naming attributes when creating data class instances

One challenge when creating instances of our data classes is keeping track of which arguments correspond to which instance attributes. In the expression `Person('David', 'Liu', 100, '40 St. George Street')`, the order of the arguments must match the order the instance attributes are listed in the definition of the data class—and it's our responsibility to remember this order. Think about how easy it would be for us to write `Person('Liu', 'David', 100, '40 St. George Street')`, only to discover much later in our program that we accidentally switched this poor fellow's given and family names![6]

[6] This is exactly the same problem with using a list to represent marriage license records.

To solve this issue, Python lets us to create data class instances using *keyword arguments* to explicitly name which argument corresponds to which instance attribute, using the exact same format as the `Person` representation we saw above:

```
>>> david = Person(given_name='David', family_name='Liu', age=100, address='40 St. George Stree
```

Not only is this more explicit, but using keyword arguments allows us to pass the argument values in any order we want:

```
>>> david = Person(family_name='Liu', given_name='David', address='40 St. George Street', age=1
```

This is a great improvement for the readability of our code when we use data classes, especially as they grow larger. One potential downside that comes with this (and in general when writing more explicit code) is that this requires more typing, and makes our code a little longer. You can get around the first issue by using auto-completion features (e.g., in PyCharm), and for the second issue you can put the different arguments on separate lines:

```
>>> david = Person(
...     family_name='Liu',
...     given_name='David',
...     address='40 St. George Street',
...     age=100
... )
```

Representing data classes in the memory model

Now that we have the ability to define our own data types, we need to decide how these data types will fit into our memory model. We'll do this by using the representation that Python displays, formatted to show each instance attribute on a new line. For example, we would represent the `david` variable in a memory model as follows:

Variable	*Value*

`david`	`Person(`
	`  family_name='Liu',`
	`  given_name='David',`
	`  address='40 St. George Street',`
	`  age=100`
	`)`