5.3 Defining Our Own Data Types, Part 2

In the previous section, we learned about *data classes*, a way to define our own data types in Python. In this section, we're going to learn study some more details about defining and designing data classes in our programs, and apply what we've learned to simplify some of work we did with tabular data in 5.1 Tabular Data.

Before we begin, please take a moment to review the `Person` data class we developed in the previous section.

```
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
```

Constraining data class values: representation invariants

In our `Person` data class definition, we specify the type of each instance attribute. By doing so, we constrain the possible values can be stored for these attributes: for example, a `Person`'s `given_name` can't be `3.5`, and `age` can't be `'millennial'`.

However, just as we saw with function type contracts, we don't always want to allow every possible value of a given type for an attribute. For example, the `age` attribute for `Person` has a type annotation `int`, but we certainly would not allow negative integers to be stored here! Somehow, we'd like to record a second piece of information about this attribute: that `age >= 0`. This kind of constraint is called a **representation invariant**, which is a predicate describing how we *represent* values of a data class. These constraints must always be true, for every instace of the data class we create—they can *never vary*.[1]

[1] The term *invariant* is used in a few different contexts in computer science; we'll explore one other kind of invariant a bit later in this chapter.

All attribute type annotations, like `age: int`, are representation invariants. However, we can express general representation invariants as well by adding them to the data class docstring. Whenever possible, we write representation invariants as Python expressions rather than English, for a reason we'll see below. Here is how we add non-type-annotation representation invariants in a class docstring:

```
@dataclass
class Person:
    """A custom data type that represents data for a perso

    Representation Invariants:
      - self.age >= 0
    """
    given_name: str
    family_name: str
    age: int
    address: str
```

One oddity with this definition is that we use `self.age` instead of `age` to refer to the instance attribute. This mimics how we access data type attributes using dot notation:

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

In the class docstring, we use the variable name `self` to refer to a generic instance of the data class.[2] This use of `self` is a strong Python convention, and we'll return to other uses of `self` later on in this course.

[2] Keep in mind that `self` here is used just in the class docstring. In the above example, the variable `david` would appear in our memory model, but `self` would not.

Checking representation invariants automatically with `python_ta`

Just as we saw with preconditions in 4.1 Specifying What a Function Should Do, representation invariants are useful parts of documentation for how a data class should be used. Like preconditions, representation invariants are *assumptions* that we make about values of a data type; for example, we can assume that every `Person` instance has an `age` that's greater than or equal to zero.

Representation invariants are also *constraints* on how we can create a data class instance. Because it can be easy to miss or ignore a representation invariant buried in a class docstring, PythonTA supports checking all representation invariants, just like it does with preconditions! Let's add a `python_ta.contracts.check_contracts` decorator to our `Person` example:

```
from dataclasses import dataclass
from python_ta.contracts import check_contracts


@check_contracts
@dataclass
class Person:
    """A person with some basic demographic information.

    Representation Invariants:
      - self.age >= 0
    """
    given_name: str
    family_name: str
    age: int
    address: str
```

If we run the above file in the Python console, we'll obtain an error whenever we attempt to create a `Person` value with invalid attributes.

```
>>> david = Person(
...     given_name='David',
...     family_name='Liu',
...     age=-100,
...     address='40 St. George Street')
Traceback (most recent call last):
  File "<input>", line 1, in <module>
  ...
AssertionError: Representation invariant "self.age >= 0" violated.
```

Notes about using `check_contracts` with data classes:

- `python_ta` is strict with the header `Representation Invariants:`. In particular, both the "`Representation`" and "`Invariants`" must be capitalized (and spelled correctly). Please watch out for this, as otherwise any representation invariants you add will not be checked!
- The `@check_contracts` decorator must be written *above* the `@dataclass` decorator. This is a current limitation of PythonTA, though we hope to improve that in the future.

The Data Class Design Recipe

Just as how functions give us a way of organizing blocks of code to represent a computation, data classes give us a way of organizing pieces of data to represent an entity. In 2.7 The Function Design Recipe, we learned a structured approach to designing and implementing functions. There is an analogous **Data Class Design Recipe**, which you should use every time you want to create a new data type for a program.[3]

[3] Note the similarities between the two recipes, such as the importance of naming and documentation.

1. Write the class header.

The class header consists of three parts: the `@dataclass` decorator (don't forget to import from `dataclasses`), the keyword `class`, and the name of the data class. Pick a short noun or noun phrase as the name of the class. The name of the class should use the "CamelCase" naming convention: capitalize every word of the class name, and do *not* separate the words with underscores.

```
@dataclass
class Person:
```

2. Write the instance attributes for the data class.

Decide on what attributes you want the data class to bundle together. Remember that every instance of the data class will have *all* of these attributes.

Each attribute name should be a short noun or noun phrase, using "snake_case" (like function and variable names). Write each annotation name and its type indented within the data class body.

```
@dataclass
class Person:
    given_name: str
    family_name: str
    age: int
    address: str
```

3. Write the data class docstring.

Create a class docstring using triple-quotes, using the same format as function docstrings. Inside the docstring, write a description of the class and a description for every instance attribute. The class description should start with a one-line summary, and you can add a longer description underneath if necessary.

Use the header "Instance Attributes:" to mark the beginning of the attribute descriptions.

```
@dataclass
class Person:
    """A data class representing a person.

    Instance Attributes:
      - given_name: the person's given nam
      - family_name: the person's family n
      - age: the person's age
      - address: the person's address
    """
    given_name: str
    family_name: str
    age: int
    address: str
```

4. Write an example instance (optional).

At the bottom of the class docstring, write a doctest example of a typical instance of the data class. This should be used to illustrate all of the instance attributes, which is especially important when the instance attributes are complex types.

```
@dataclass
class Person:
    """A data class representing a person.

    Instance Attributes:
      - given_name: the person's given nam
      - family_name: the person's family n
      - age: the person's age
      - address: the person's address

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

5. Document any additional representation invariants.

If there are representation invariants for the instance attributes beyond the type annotations, include them in the class docstring under a separate section "Representation Invariants:" in between the instance attribute descriptions and sample instance.

Just as with function preconditions, each representation invariant should be a boolean expression in Python. Use `self.<attribute>` to refer to an instance attribute within a representation invariant.

```
@dataclass
class Person:
    """A data class representing a person.

    Instance Attributes:
      - given_name: the person's given nam
      - family_name: the person's family n
      - age: the person's age
      - address: the person's address

    Representation Invariants:
      - self.age >= 0

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

A worked example

To wrap up our introduction of data classes, let's see how to apply data classes to the marriage license data set we studied in 5.1 Tabular Data.

ID	Civic Centre	Marriage Licenses Issued	Time Period
1657	ET	80	January 1, 2011
1658	NY	136	January 1, 2011
1659	SC	159	January 1, 2011
1660	TO	367	January 1, 2011
1661	ET	109	February 1, 2011
1662	NY	150	February 1, 2011
1663	SC	154	February 1, 2011
1664	TO	383	February 1, 2011

Recall that we represented the data as a list of lists:

```
>>> marriage_data = [
...     [1657, 'ET', 80, datetime.date(2011, 1, 1)],
...     [1658, 'NY', 136, datetime.date(2011, 1, 1)],
...     [1659, 'SC', 159, datetime.date(2011, 1, 1)],
...     [1660, 'TO', 367, datetime.date(2011, 1, 1)],
...     [1661, 'ET', 109, datetime.date(2011, 2, 1)],
...     [1662, 'NY', 150, datetime.date(2011, 2, 1)],
...     [1663, 'SC', 154, datetime.date(2011, 2, 1)],
...     [1664, 'TO', 383, datetime.date(2011, 2, 1)]
... ]
```

We implemented the following function to calculate the average number of marriage licenses issued by a particular civic centre:

```
def average_licenses_issued(data: list[list], civic_centre: str) -> float:
    """Return the average number of marriage licenses issued by civic_centre in data.

    Precondition:
      - all({len(row) == 4 for row in data})
      - any({row[1] == civic_centre for row in data})
    """
    issued_by_civic_centre = [row[2] for row in data if row[1] == civic_centre]

    total = sum(issued_by_civic_centre)
    count = len(issued_by_civic_centre)

    return total / count
```

Here is how we will use data classes to simplify this approach. Rather than storing each row in the table as a list, we can instead introduce a new data class to store this information:

```
from dataclasses import dataclass
import datetime


@dataclass
class MarriageData:
    """A record of the number of marriage licenses issued in a civic centre in a given month.

    Instance Attributes:
      - id: a unique identifier for the record
      - civic_centre: the name of the civic centre
      - num_licenses: the number of licenses issued
      - month: the month these licenses were issued
    """
    id: int
    civic_centre: str
    num_licenses: int
    month: datetime.date
```

Then using this data class, we can represent tabular data as a list of `MarriageData` instances rather than a list of lists. Not much has changed! The values representing each entry in the table are the same, but how we "bundle" each row of data into a single entity is different.

```
>>> marriage_data = [
...     MarriageData(1657, 'ET', 80, datetime.date(2011, 1, 1)),
...     MarriageData(1658, 'NY', 136, datetime.date(2011, 1, 1)),
...     MarriageData(1659, 'SC', 159, datetime.date(2011, 1, 1)),
...     MarriageData(1660, 'TO', 367, datetime.date(2011, 1, 1)),
...     MarriageData(1661, 'ET', 109, datetime.date(2011, 2, 1)),
...     MarriageData(1662, 'NY', 150, datetime.date(2011, 2, 1)),
...     MarriageData(1663, 'SC', 154, datetime.date(2011, 2, 1)),
...     MarriageData(1664, 'TO', 383, datetime.date(2011, 2, 1))
... ]
```

And here is how we could modify our `average_licenses_issued` function.

```
def average_licenses_issued(data: list[MarriageData], civic_centre: str) -> float:
    """Return the average number of marriage licenses issued by civic_centre in data.

    Precondition:
      - any({row.civic_centre == civic_centre for row in data})
    """
    issued_by_civic_centre = [
      row.num_licenses for row in data if row.civic_centre == civic_centre
    ]

    total = sum(issued_by_civic_centre)
    count = len(issued_by_civic_centre)

    return total / count
```

Again, not much has changed: instead of writing `row[1]` and `row[2]`, we instead write `row.civic_centre` and `row.num_licenses`. This is longer to write, but also more explicit in what attributes of the data are accessed. And to quote from the Zen of Python, *explicit is better than implicit*.

Summary: why data classes?

Last section, we claimed that a data class is a better way of representing a bundle of data than a list. Let's review a few reasons why:

1. We now access the different attributes by name rather than list index, which is easier to remember and understand if you're reading the code.
2. Similarly, software like PyCharm and `python_ta` understand data class definitions, and will warn us if we try to create *invalid person values* (e.g., wrong arguments to `Person`), or access invalid attributes.
3. Lists are designed to be a very flexible and general data type, and support many operations (e.g. list concatenation and `in`) that we don't want to do for actual people or records of marriage data. Now that we use a separate data class, we eliminate the possibility of using these list operations on a "marriage data row", even accidentally.