Module 8: Classes

Exercise

If you have not already, get prepared for class by downloading the start code:

```
!wget https://student.cs.uwaterloo.ca/~cs114/src/module-08-start.ipynb
```

Storing complex data

We now have several ways to store data:

- We use **lists**, especially to store data of variable length, all of the same type;
- we use **dictionaries**, to store keys and associated values;
- we use **tuples** to store fixed-size unchangeable data.

Consider: I want to store information about a "country". I want to store 3 things: the **continent** (a `str`), the **name** of the leader (a `str`), and the **population** (an `int`). How can we do it?

Storing complex data as a list

```
canada = ["North America", "Trudeau", 38526760]
india = ["Asia", "Modi", 1352642280]
```

- 👍 We can change the values.
- 👎 Nothing helps us remember what kind of things we're supposed to be storing.
- 👎 Nothing helps us avoid corrupting the data, or even keeping it the right size; we can add/remove values with `.append` and `.pop`.
- 👎 How do I annotate this type?

Storing complex data as a tuple

```
canada = ("North America", "Trudeau", 38526760)
india = ("Asia", "Modi", 1352642280)
```

- 👎 Nothing helps us remember what kind of things we're supposed to be storing.
- 👍 The size is fixed, so we can't accidentally add or remove data.
- 👎 We can't change anything at all!
- 👍 How do I annotate this type? Like `tuple[str, str, int]`. That's not bad.

Storing complex data as a dict

```
canada = {
    "continent": "North America",
    "leader": "Trudeau",
    "population": 38526760
}
india = {
    "continent": "Asia",
    "leader": "Modi",
    "population": 1352642280
}
```

- 👍 The named fields help us remember what each piece is for.
- 👍 We can still change the values.
- 👎 Nothing helps us avoid corrupting the data, or even keeping it the right size.
- 👎 How do I annotate this type?

Storing complex data: lots of reasonable options

It can be quite appropriate to use a `list`, a `dict`, or a `tuple` to store data. Different structures are better in different circumstances.

Often, the tools we have just summarized are a good choice.

A big part of programming is figuring out how to store whatever it is you need to store.

`list`? `dict`? `tuple`? A combination? Something else?

I can't give you simple rules of how to choose which. You'll develop intuition with experience. CS115/CS135 will help; see especially CS234 "Data Types and Structures" or CS240 "Data Structures and Data Management" (if you become a CS major).

For now, we will see a taste of an alternative that gives us certain neat features: **classes**.

Classes

A **class** is the way we create a **new type**.

Write the keyword `class` followed by the name of the class, a colon, and a block of code.

```
class Country:
    """Describe a country."""
    continent: str
    leader: str
    population: int
```

We use a **docstring** and **annotations** to help other programmers see what this type is for.

This does nothing. It just documents our class, indicating that our `Country` is supposed to store these three **attributes** with these types.

...It means we can call `help(Country)` and get a message that is at least slightly useful.

> **!** Write a docstring and attribute annotations for every class you create!

Storing data in a `class`

The rest of what we do in a `class` statement is define functions.

Functions defined inside a class are called **methods**.

The first method we need to create is a **magic method** that creates an object of this type.

Here's what it looks like for our `Country` class:

```
class Country:
    def __init__(self, continent: str, leader: str, population: int) -> None:
        self.continent = continent
        self.leader = leader
        self.population = population
```

We don't need use the same names for the parameters of `__init__` as for the attributes. But we can.

Storing data in a `class`

```
class Country:
    def __init__(self, continent: str, leader: str, population: int) -> None:
        self.continent = continent
        self.leader = leader
        self.population = population
```

In classes, following tradition, we will always name the first parameter `self`.

This first parameter refers to the object we are working in; for `__init__`, it refers to a new object that we are creating. When we call the class, it creates a new object for `self`, then calls this `__init__` function.

```
canada = Country("North America", "Trudeau", 36524723)
```

Notice only 3 parameters. The first parameter, `self`, is created automatically.

This code assigns values to certain **attributes** of the newly created object. So now:

```
canada.continent ⇒ "North America"
canada.population ⇒ 36524723
```

It's kinda like a `dict[str, any]`, using a different syntax.

Documenting and annotating a `class`

1. Write a docstring for the class.
2. Annotate all the attributes, by listing each followed by colon, space, and the type.
3. Don't annotate `self`. Annotate the rest of the parameters.
4. The `__init__` method always returns `None`.
5. **Magic** methods have implicit purpose, so we may omit the docstring for them.

```
class Country:
    """Describe a country."""
    continent: str
    leader: str
    population: int
    def __init__(self, continent: str, leader: str, population: int) -> None:
        self.continent = continent
        self.leader = leader
        self.population = population
```

Practice: Creating a `class`

```
class Country:
    """Describe a country."""
    continent: str
    leader: str
    population: int
    def __init__(self, continent: str, leader: str, population: int) -> None:
        self.continent = continent
        self.leader = leader
        self.population = population
```

Exercise

Following this model, create a fully-documented class `Hero` that stores a `name`, `year`, and `items`. For example:

```
frodo = Hero("Baggins, Frodo", 2968, ["One Ring", "Sting"])
check.expect("attributes", frodo.name, "Baggins, Frodo")
check.expect("attributes", frodo.year, 2968)
check.expect("attributes", frodo.items, ['One Ring', 'Sting'])
```

Storing complex data as a `Country`

Now that we have defined a `class`, how is it to use?

```
canada = Country("North America", "Trudeau", 36524723)
```

- We can change the values:

  ```
  canada.leader = 'Mr. Bean'
  canada.population += 1
  ```

- The class definition helps us remember what kind of things we're supposed to be storing.
- It is still possible to add or remove attributes from the class. But we're quite unlikely to use such tools by accident.
- How do I annotate this type? Like `Country`.

Printing an object so far gives something useless:

```
print(canada)
<__main__.Country object at 0x101247c50>
```

If we add a magic method `__repr__(self)`, it will call that function and print what it returns.

```
    def __repr__(self) -> str:
        return ("CNT: " + self.continent + "; L: " + self.leader
                + "; POP: " + str(self.population))
print(canada)
CNT: North America; L: Trudeau; POP: 36524723
str(canada) ⇒ "CNT: North America; L: Trudeau; POP: 36524723"
```

`__repr__` can't have any argument except `self`, which we don't annotate. It must return `str`.

Exercise

Add the `__repr__` method to your `Hero` class so it displays name and birthyear.

```
check.expect("repr:", str(frodo), 'Baggins, Frodo, born 2968')
```

Aliases

We can have lists which are aliases:

```
L = [2,4,6,0,1]
M = L
M[2] = 7
# L[2] is also 7.
```

Similarly, we have have objects which are aliases:

```
canada = Country("North America",
                 "Trudeau", 36524723)
cold_place = canada
cold_place.population = 7
canada.population ⇒ 7
```

With lists you can avoid aliasing by using `M = L.copy()`. The `list.copy` method creates a new list that contains the same values.

But by default we don't have a `.copy` method. We can do it manually:

```
cold_place = Country(canada.continent, canada.leader, canada.population)
cold_place.population = 7
# canada.population is unchanged.
```

Equality

Consider: I create two `Country` objects, in the same way:

```
canada =      Country("North America", "Trudeau", 36524723)
cold_place = Country("North America", "Trudeau", 36524723)
```

They should be identical. But nonetheless: `canada == cold_place` ⇒ **`False`**.

Since `canada` and `cold_place` seem identical, we want they to be "equal" according to the `==` operator. The magic method `__eq__(self, other)` defines what `==` means.

```
    def __eq__(self, other: any) -> bool:
        return (isinstance(other, Country)
                and self.continent == other.continent
                and self.leader == other.leader
                and self.population == other.population)
```

Now `canada == cold_place` is equivalent to `canada.__eq__(cold_place)`, and this will return **`True`**.

Equality

Let's look at this code carefully:

```
def __eq__(self, other: any) -> bool:
    return (isinstance(other, Country)
            and self.continent == other.continent
            and self.leader == other.leader
            and self.population == other.population)
```

- The built-in function `isinstance(val, t)` returns `True` if `val` is derived from class `t`.
- The rest of the code just checks that the attributes of the two objects are the same.

Exercise

Add the `__eq__` method to your `Hero` class so people born in the same year are equal:

```
hermione = Hero("Granger, Hermione", 1979, ["Time Turner"])
check.expect("==", hermione == Hero("Doe, John", 1979, []), True)
check.expect("==", hermione == Hero("Granger, Hermione", 1980, ["Time Turner"]),
    False)
check.expect("==", hermione == 1979, False)
```

Exercise

Write a function `make_countries` that takes three lists of equal length and returns a `list[Country]`.

```
check.expect("countries",
    make_countries(["Asia", "North America", "Europe", "Asia"],
                   ["Modi", "Trudeau", "Macron", "Yoon"],
                   [1339491960, 36524723, 67396432,  51745000]),
    [Country("Asia", "Modi", 1339491960),
     Country("North America", "Trudeau", 36524723),
     Country("Europe", "Macron", 67396432),
     Country("Asia", "Yoon", 51745000)])
```

We couldn't do this exercise before we created the `__eq__` magic method... Why not?

The `check.expect` function needs to be able to check equality! The `Country("North America", "Trudeau", 36524723)` created by `make_countries` won't be the exact same one we wrote inside our test!

Magic Methods for Many Purposes

Magic methods are how all sorts of things work.

We will never call magic methods directly, but any time we see code that works with objects, we will need to look at them to understand what it does.

We have seen that `a == b` calls `a.__eq__(b)`. That is, it calls the `__eq__` magic method on `a`, using `a` for **`self`**, and `b` for `other`.

We can define magic methods to specify behaviour for all operators:

- `__add__` defines the behaviour of `+`, `__sub__` of `-`, `__mul__` of `*`, `__floordiv__` of `//`, etc.
- `__getitem__` defines the behaviour of slicing; see **`help(list.__getitem__)`**. And so on. Python is fundamentally an object oriented language; all our types are “classes”.

Non-magic methods

So far, we have only define **magic** methods: methods that define how Python internals work for objects of our class.

We've also seen ordinary (non-magic) methods for lots of types:

- `list.pop` is a method that drops a value from the `list`:

  ```
  L = [2,4,6,0,1]
  L.pop(1) ⇒ 4
  L ⇒ [2,6,0,1]
  ```

- `str.split` is a method that returns a `list[str]` containing the words in the `str`:

  ```
  s = "give peas a chance"
  s.split() ⇒ ['give', 'peas', 'a', 'chance']
  ```

To create these, the `list` class would have a line like `def pop(self, index):`, and the `str` class would have a line like `def split(self):`.

Non-magic methods

We define a non-magic method in the same way as a magic method.

Usefully it might:

- mutate the object, like `list.pop`;
- have a **side effect** like printing a message, reading/writing a file, drawing a plot, etc.;
- return a value of some kind, like `str.split`.

Consider this method for our `Country` class:

```
class Country:
    def election(self, winner: str) -> None:
        """Update self when winner wins, and print a message."""
        print("Election Results:")
        if self.leader == winner:
            print(self.leader + " re-elected")
        else:
            print(winner + " replaces " + self.leader)
            self.leader = winner
```

Non-magic methods

```
class Country:
    def election(self, winner: str) -> None:
        """Update self when winner wins, and print a message."""
        print("Election Results:")
        if self.leader == winner:
            print(self.leader + " re-elected")
        else:
            print(winner + " replaces " + self.leader)
            self.leader = winner
```

Suppose we have the following:

```
usa = Country("North America", "Trump", 329531886)
usa.leader ⇒ "Trump"
usa.election("Biden") # The method mutates, prints, and returns None.
usa.leader ⇒ "Biden"
```

`usa.election("Biden")` call the `Country.election` method, with **`self`** being the `usa` object, and `winner` being `"Biden"`.

Non-magic method

```
class Country:
    def election(self, winner: str) -> None:
        """Update self when winner wins, and print a message."""
        print("Election Results:")
        if self.leader == winner:
            print(self.leader + " re-elected")
        else:
            print(winner + " replaces " + self.leader)
            self.leader = winner
```

Exercise

Using this code as a model, add to your `Hero` class a method `take(item)` that takes a `str`, mutates the object to add `item` to the list of items it holds, and returns `None`.

```
hermione = Hero("Granger, Hermione", 1979, ["Time Turner"])
check.expect("take returns None", hermione.take("Horcrux"), None)
check.expect("items was mutated", hermione.items, ['Time Turner', 'Horcrux'])
```

Exercise: `list[Country]`

Exercise

Write a function `reslice`. It takes a `list[Country]` and returns a `tuple[list[str], list[str], list[int]]`, containing the continents, the leaders, and the populations.

(This is like the inverse of `make_countries` we wrote earlier.)

```
check.expect("reslice",
             reslice([Country("Asia", "Modi", 1339491960),
                      Country("North America", "Trudeau", 36524723),
                      Country("Europe", "Macron", 67396432),
                      Country("Asia", "Yoon", 51745000)]),
             (["Asia", "North America", "Europe", "Asia"],
              ["Modi", "Trudeau", "Macron", "Yoon"],
              [1339491960, 36524723, 67396432,  51745000]))
```

Module summary

- Understand how to group information into a single object as a `class`.
- Know how to **document** and **annotate** classes.
- Be able to create **magic methods** to initialize objects, and to define the behaviour of operators such as == and +.
- Be able to create (non-magic) methods to add capabilities to classes.

Before we begin the next module:

- Read and complete the exercises in module 8 of the online textbook, at https://online.cs.uwaterloo.ca/
- Complete the module 8 Review Quiz, due on Monday.