Object-Oriented Programming Vocabulary

From http://interactivepython.org/courselib/static/thinkcspy/ClassesBasics/Glossary.html

- **class**: A user-defined compound type. A class can also be thought of as a template for the objects that are instances of it.
- **attribute**: One of the named data items that makes up an instance.
- **method**: A function that is defined inside a class definition and is invoked on instances of that class.
- **initializer (or constructor) method**: A special method in Python (called `__init__`) that is invoked automatically to set a newly-created object’s attributes to their initial state.
Object-Oriented Programming Vocabulary

From http://interactivepython.org/courselib/static/thinkcspy/ClassesBasics/Glossary.html

▶ **object**: A compound data type that is often used to model a thing or concept in the real world. It bundles together the data and the operations that are relevant for that kind of data. Instance and object are used interchangeably.

▶ **instance**: An object whose type is of some class. Instance and object are used interchangeably.

▶ **to instantiate**: To create an instance of a class, and to run its initializer.

▶ **object-oriented programming**: A powerful style of programming in which data and the operations that manipulate it are organized into classes and methods.

▶ **object-oriented language**: A language that provides features, such as user-defined classes and inheritance, that facilitate object-oriented programming.
The \_\_str\_\_(self) method

It is often useful to define how an object of given class should be converted to a string (e.g. for printed). This is achieved by defining the method \_\_str\_\_(self):

```python
def \_\_str\_\_(self):
    """
    Args: Self,
    Returns: String describing bus
    """
    return "Bus at station \"{}\" contains passengers \"{}\"".format(self.station, self.passengers)
```

Then:
my\_bus = Bus()
print(my\_bus) \# will execute \_\_str\_\_() on my\_bus to get a String, which then gets printed.
Putting it all together

See busSim_object_oriented.py

Notice how much simpler the simulation loop becomes!

Advantage: All the code that pertains to the bus behavior is in the Bus class. The programmer of the simulation loop does not need to know all the details of the Bus class. It only needs to know how to use its methods properly.
Revisiting our medical_diagnostic program

Our program was a bit complicated because data and code pertaining to different concepts are intermingled.

▶ **Symptoms**
  - Data: Symptoms present and absent were stored in a tuple. Programmer needs to remember that the first element of the tuple corresponds to the symptoms that are present, and the second to the symptoms that are absent.
  - Code: `symptom_similarity` function

▶ **Patients**
  - Data: Patients' symptoms and diagnostics were stored in separate dictionaries: `all_patients_symptoms`, `all_patients_diagnostics`
  - Code: `most_similar_patients()`, `diagnostics_from_symptoms()`, `recommend_symptom_to_test()`

▶ **Probabilistic diagnostics**
  - Data: Dictionary of diseases with associated probabilities.
  - Code: `count_diagnostics()`, `pretty_print_diagnostics()`, `diagnostic_clarity()`:
An object-oriented medical_diagnostic program

Idea: Define separate classes for separate concepts:

- Symptoms
- Patient
- Probabilistic diagnostics

Each class will be stored in a different Python file (also called a module): symptoms.py, patient.py, probabilistic_diagnostic.py.

A module can import code (classes, functions, etc.) from another module.

This allows big programs to be broken down into smaller, more digestible chunks.

- Makes easier understanding, developing, and debugging large programs
Symptoms class

- Attributes:
  - present: Set of symptoms (Strings) that are present
  - absent: Set of symptoms (Strings) that are absent

- Methods:
  - `__init__`(self, pres, abs)
  - `symptom_similarity`(self, other)
  - `__str__`(self)

See symptoms.py
Probabilistic_diagnostic class

- **Attributes:**
  - `prob`: Dictionary of diagnostic probabilities
  - `symptoms`: Object of class Symptoms
  - `diagnostic`: String

- **Methods:**
  - `__init__(self)`
  - `count_diagnostics(self, patient_set)`
  - `pretty_print_diagnostics(self)`
  - `diagnostic_clarity(self)`

See `probabilistic_diagnostic.py`
Patient class

▶ Attributes:
  ▶ ID: Integer
  ▶ symptoms: Object of class Symptoms
  ▶ diagnostic: String

▶ Methods:
  ▶ __init__(self, my_patient_ID, my_symptoms, my_diagnostic)
  ▶ most_similar_patients(self, all_patients, n_top=10)
  ▶ diagnostics_from_symptoms(self, all_patients, n_top=10)
  ▶ recommend_symptom_to_test(self, all_patients, n_top=10)
  ▶ __str__(self)

See patient.py

Note: The Patient class needs to know about the Symptoms and Probabilistic_diagnostic classes. So:

```python
# import the class Symptoms from file symptoms.py
from symptoms import Symptoms

# import the class Probabilistic_diagnostic from file probabilistic_diagnostic.py
from probabilistic_diagnostic import Probabilistic_diagnostic
```
Our code that puts everything together is in a separate file: medical_diagnostic_tester.py.

It needs to import the three other modules:

```python
1 from symptoms import Symptoms
2 from patient import Patient
3 from probabilistic_diagnostic import Probabilistic_diagnostic
```