Instructions to Execute Code

This code only consists of classes. Classes may be accessed from the Python console, or from importing the system_implementation module.

Commentary on Approach

Because this assignment is for a class diagram implementation, it was important to limit the scope of the code development. A consultant must not take liberties with a client's requirements unless absolutely necessary (Buckley, 2022).

The goal of this code is to reflect the provided class diagram, and it does so with one minor exception. To establish the specified multiplicity for Patient objects registered to Doctor objects, a registered_patient_list was created in the Doctor class with a register_patient method that stores a maximum of 500 Patient objects in each Doctor's registered_patient_list.

For some methods, the way the surgery conducts its business practices – such as requesting appointments, requesting repeat prescriptions, and conducting consultations – are unknown. In these cases, the resultant action is to use the data from the involved objects to create a "print" statement that describes the resultant action.

This code is documented with docstrings, explained with comments, and follows best practices such as duck typing, self-documenting code, Python Enhancement Proposal (PEP) 8: the Style Guide for Python, and PEP 257: Docstring Conventions.

Description of Design

This design mirrors the class diagram, with all corresponding attributes and methods contained within each class. The Doctor and Nurse classes inherit from the Healthcare Professional class, Appointment Schedule is composed of Appointments, and both Patients and Doctors aggregate Appointments and Prescriptions.

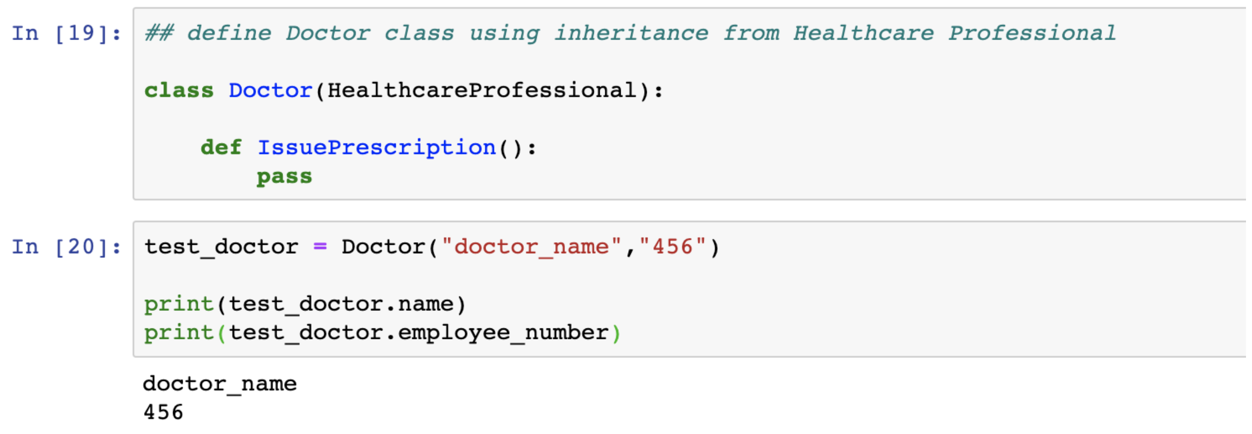
Reflection on Development Process

The development process began by defining each of the classes and attributes, and initially declaring methods with a "pass" statement.

Method actions were designed to provide rudimentary functionality, given the unknowns on the overall user or application interface that will interact with this class structure. In particular, the Appointment Schedule was developed with a dictionary to use appointment dates and times as keys, assigning Appointment objects as values. Sample times were loaded into the object's initialisation to enable development and testing. Where pragmatic, methods check to make sure that Patient and Prescription data match, or that an Appointment Schedule time is valid and available. Functions such as try, catch, and exception were not implemented due to interface unknowns.

Testing

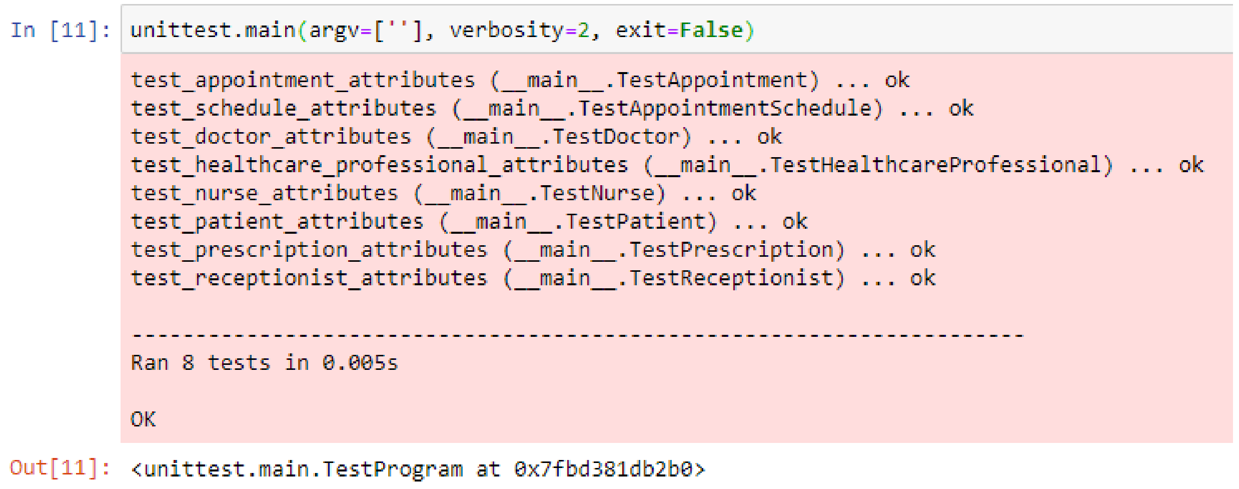
Manual, or exploratory, testing began after the definition of each class and verified proper instantiation of class and super-class attributes (Shaw, n.d.). A sample follows below:



```
In [19]: ## define Doctor class using inheritance from Healthcare Professional

         class Doctor(HealthcareProfessional):

             def IssuePrescription():
                 pass

In [20]: test_doctor = Doctor("doctor_name","456")

         print(test_doctor.name)
         print(test_doctor.employee_number)

         doctor_name
         456
```

Unit Testing

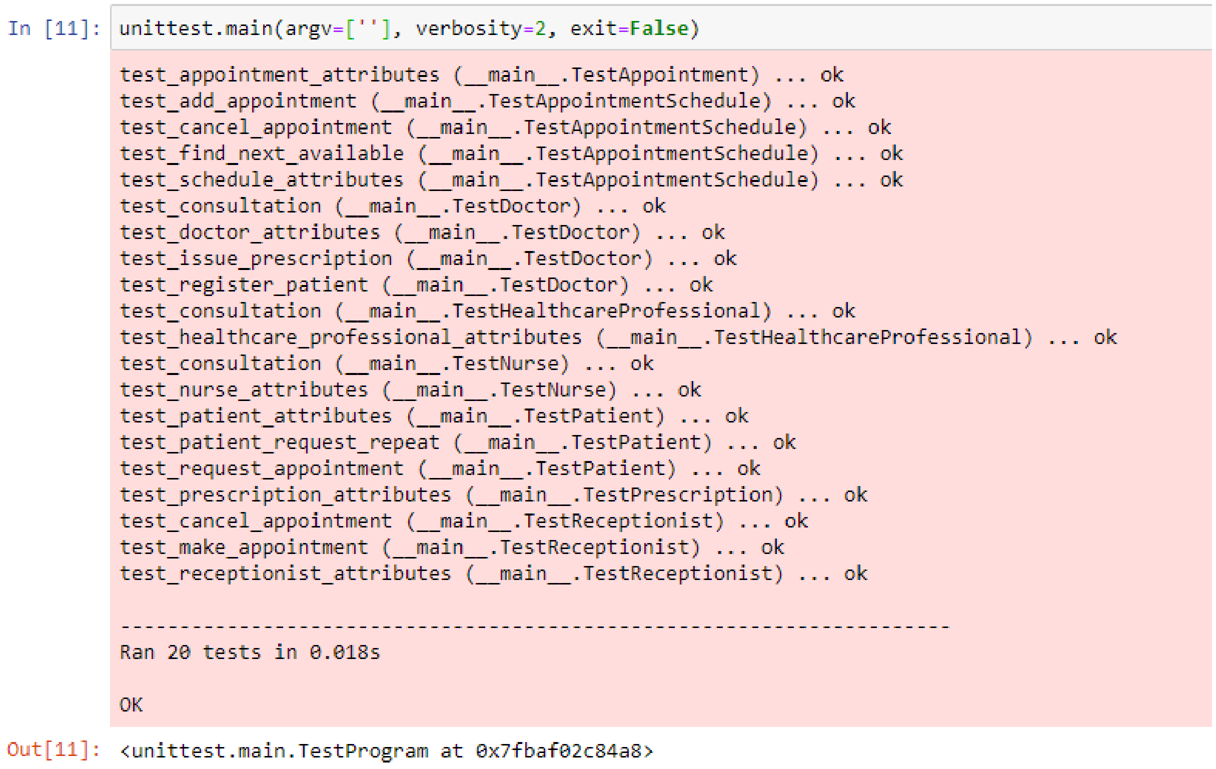
After the classes, attributes and methods were defined, a unit testing script using the "unittest" module in Python was developed in a separate test script file (Sabato, 2021). This automated the testing of class instantiation, attributes, inherited features, and methods that did not rely on external variables (Shaw, n.d.). Some methods required verifying the accuracy of print outputs using the "mock" and "patch" functions from the "unittest" module (Trudeau, n.d.; Python Software Foundation, 2022). The unit testing output follows below.



```
In [11]: unittest.main(argv=[''], verbosity=2, exit=False)

         test_appointment_attributes (__main__.TestAppointment) ... ok
         test_schedule_attributes (__main__.TestAppointmentSchedule) ... ok
         test_doctor_attributes (__main__.TestDoctor) ... ok
         test_healthcare_professional_attributes (__main__.TestHealthcareProfessional) ... ok
         test_nurse_attributes (__main__.TestNurse) ... ok
         test_patient_attributes (__main__.TestPatient) ... ok
         test_prescription_attributes (__main__.TestPrescription) ... ok
         test_receptionist_attributes (__main__.TestReceptionist) ... ok

         ----------------------------------------------------------------------
         Ran 8 tests in 0.005s

         OK

Out[11]: <unittest.main.TestProgram at 0x7fbd381db2b0>
```

Integration Testing

Once unit tests complete and functional, mock objects were instantiated to test interactions between objects. Whilst integration testing is often separated from unit testing when working with external databases and application programming interfaces, it was more pragmatic to combine the two in a single test since all elements of this test originate from the same Python module (Shaw, n.d.). This required incorporating different functions from “unittest” to determine if appointment objects had been appropriately stored and removed from the schedule dictionary in the Appointment_Schedule. The integration test result follows below.



```
In [11]: unittest.main(argv=[''], verbosity=2, exit=False)

test_appointment_attributes (__main__.TestAppointment) ... ok
test_add_appointment (__main__.TestAppointmentSchedule) ... ok
test_cancel_appointment (__main__.TestAppointmentSchedule) ... ok
test_find_next_available (__main__.TestAppointmentSchedule) ... ok
test_schedule_attributes (__main__.TestAppointmentSchedule) ... ok
test_consultation (__main__.TestDoctor) ... ok
test_doctor_attributes (__main__.TestDoctor) ... ok
test_issue_prescription (__main__.TestDoctor) ... ok
test_register_patient (__main__.TestDoctor) ... ok
test_consultation (__main__.TestHealthcareProfessional) ... ok
test_healthcare_professional_attributes (__main__.TestHealthcareProfessional) ... ok
test_consultation (__main__.TestNurse) ... ok
test_nurse_attributes (__main__.TestNurse) ... ok
test_patient_attributes (__main__.TestPatient) ... ok
test_patient_request_repeat (__main__.TestPatient) ... ok
test_request_appointment (__main__.TestPatient) ... ok
test_prescription_attributes (__main__.TestPrescription) ... ok
test_cancel_appointment (__main__.TestReceptionist) ... ok
test_make_appointment (__main__.TestReceptionist) ... ok
test_receptionist_attributes (__main__.TestReceptionist) ... ok

----------------------------------------------------------------------
Ran 20 tests in 0.018s

OK

Out[11]: <unittest.main.TestProgram at 0x7fbaf02c84a8>
```

System Testing

System testing followed the end-to-end scenarios for an Appointment object: a Patient object requested an Appointment, a Receptionist made the appointment, the Appointment Schedule added the appointment, a Nurse conducted the Consultation, the Appointment Schedule found the next appointment available, and the Receptionist cancelled the Appointment.

References

Buckley, O. (2022) *Seminar 5: System Implementation Discussion* [Zoom]. OOIS_PCOM7E March 2022 A Object-oriented Information Systems March 2022. University of Essex Online. [Accessed 11 May 2022].

Python Software Foundation. (2022). *unittest — Unit testing framework — Python 3.8.2 documentation*. [online] Available at: https://docs.python.org/3/library/unittest.html. [Accessed 11 May 2022].

Sabato, C. (2021). *How To Write a Unit Test in Python: A Simple Guide*. [online] CODEFATHER. Available at: https://codefather.tech/blog/write-unit-test-python/. [Accessed 21 May 2022].

Shaw, A. (n.d.). *Getting Started With Testing in Python – Real Python*. [online] realpython.com. Available at: https://realpython.com/python-testing/. [Accessed 21 May 2022].

Trudeau, C. (n.d.). *Mocking print() in Unit Tests – Real Python*. [online] realpython.com. Available at: https://realpython.com/lessons/mocking-print-unit-tests/. [Accessed 21 May 2022].