CS5710 :- Lab 4

Stephen Piddock

February 7, 2024

1 Learning outcomes

This lab session will introduce you to more advanced concepts in object-oriented programming such as

- inheritance,
- method overloading,
- polymorphism.

2 Notes

- Complete the various sections described in this document. and have a look at the extensions.
- **Remember** the TAs and lecturers are here to help - if you are stuck then don't hesitate to ask questions.
- Read carefully the instructions, and ask for help if you feel lost.
- It is recommended that you store your programs in a folder hierarchy comprising of a single high-level folder, e.g., `CS5710Labs`, and one sub-folder for each lab session, e.g., `lab2`, `lab3`, etc.
- Unless stated otherwise, store your programs in the files called `ex<exercise_number>.py`. For example, the program for Exercise 1 should be stored in file `ex1.py`.
- If you work on a lab workstation, use the `Y:` folder to store your work as everything stored in the local folders is being erased on a daily basis.

- If your program gets stuck in an infinite loop, you can interrupt it with `CTRL-C` or click on the red box above the console window in Spyder.

3 Creating and testing a base class

3.1 Create a base class `Person` inheriting from the Python's `object` class, and add code to its methods replacing the `pass` keywords in the following template (available on Moodle). Save your code in a file named `Person.py` in your current folder.

```
from datetime import date

class Person(object):

    def __init__(self, name):
        """
        name, string
        birthday, instance of datetime.date
        Creates a person setting self's name attribute to
        the value of the name argument, and birthday to None.
        """
        pass

    def setBirthday(self, birthday):
        """
        birthday, an instance of datetime.date
        Sets self's birthday attribute to birthday
        """
        pass

    def getBirthday(self):
        """
        Returns self's birthday attribute
        """
        pass

    def getName(self):
        """
        Returns self's name attribute
        """
```

```
        pass

    def getAge(self):
        """
        Returns self's current age in years.
        Hint: Use date.today().year to get the year of today, and
        the year attribute of birthday to get the birthday's
        year. Return the difference between the two.
        """
        pass

    def __lt__(self, other):
        """
        Returns True if self's name is lexicographically
        less than other's name, and False otherwise.
        Implementing this method will enable sorting of lists
        comprised of instances of this class.
        """
        pass

    def __str__(self):
        """
        Returns self's string representation
        combining the values of self's name and
        birthday attributes.
        A possible return value could look like
        '(Person:<self.name>:<self.birthday>)'
        as in '(Person:John Doe:1999-09-09)'.
        """
        pass
```

3.2 Follow the instructions below to test your `Person` class:

- Add a conditional statement `if __name__ == '__main__':` at the end of `Person.py`. The `if`'s condition will only evaluate to `True` if `Person.py` is executed as a standalone script. It will be `False` if the `Person` module is *imported* into the main scope of another Python program.

  This is an important coding idiom to adopt as it allows your

classes to be imported by other programs without causing them to automatically execute their testing code!

- Add some testing code in the indented code block following the `if` statement above. Create a few instances of `Person`, invoke its methods on those instances, and print out the results. Here is an example of what you can test:

```
if __name__ == '__main__':
    # Create an instance of Person
    p = Person("Tom Cruise")

    # Set birthday attribute
    # Tom Cruise was born on 3 July 1962
    p.setBirthday(date(1962, 7, 3))

    # Outputs the string representation of p
    # as determined by your __str__() method code
    print(p)

    # Tom Cruise is 56 years old
    print(p.getName(), "is", p.getAge(), "years old")

    # Create a list of Person objects initialized with the give
    # names and default birthdays
    plist = [Person("John Doe"), Person("Jane Smith"),
                 Person("Sheldon Cooper"),
             Person("Jason Bourne"), Person("Anna Gillian")]

    # Sorts list in the order consistent with <
    # as determined by your __lt__() method logic.
    # Sorting of a list (or any other iterable type)
    # is automatically enabled provided its instances
    # implement the __lt__() method.
    plist.sort()

    # Uses list comprehension (for/in inside list brackets [])
    # to create a list of string representations
    # out of the Person objects stored in plist,
    # and prints the resulting list.
    # This is more compact than printing in a loop though
```

```
# not as efficient since a new copy of the list is created.
str_list = [str(p) for p in plist]

# Should output a list of stringified Person instances
# sorted in the lexicographical order of their names.
print(str_list)
```

4 Checking object types

Built-in functions `isinstance()` and `type()` are useful for querying the type of a given object. They are defined as follows:

- `isinstance()` takes an object and a type as arguments, and returns `True` if the object is an instance of the given type, and `False`, otherwise. For example, if a variable `a` is bound to an instance of `Animal`, then `isinstance(a, Animal)` will return `True`.
- `type()` takes an object and returns its type (i.e., an object representing the type, not a string!), which can then be compared for equality against another type. For example, if a variable `a` is bound to an instance of `Animal`, then `type(a) == Animal` will evaluate to `True`.

The two functions are not exactly the same: `isinstance()` will correctly identify instances of a superclass whereas `type()` will always return the exact type of its argument. For example, if `d` is bound to an instance of `Dog`, which inherits from `Animal`, then `isinstance(d, Animal)` will return `True` whereas `type(d) == Animal` will evaluate to `False`.

1. Add code to the testing section of `Person.py` that binds a variable to an instance of `Person`, tests if it is an instance of `Person`, using both `isinstance()` and `type()`, and outputs the results of both tests.
2. Add code to the testing section of `Person.py` that binds a variable to an instance of `Person`, tests if it is an instance of `object` (which is the superclass of `Person`) using `isinstance()`, and outputs the result. Try to do the same by comparing the return value of `type()` to `object`. Verify that the output is `False` in this case as `type()` is oblivious to subclassing.

   `isinstance()` can be used to make your code more robust. For example, you can test if an argument of a method or a function is of

expected type and raise an error if it is not. This is a good programming practice to adopt when coding in a *dynamically-typed* language, such as Python.

3. Modify the code of the method `setBirthday()` of `Person` so that it only sets `self.birthday` if its `birthday` argument is bound to an instance of `date`. Otherwise, it raises `TypeError` using either `raise TypeError()`, or `raise TypeError('<message string>')`.

   Raising an error has an effect of interrupting the program execution, and unwinding the function invocation sequence up to the first occurrence of an *error handling code*. If no such code was provided, the error will propagate all the way up to the main scope causing the program to prematurely terminate with an error message being output on the screen.

4. In the testing section of `Person.py`, test your modified implementation of `setBirthday()` by calling it with an argument of an incorrect type as in the following example:

```
x = Person("Eve Polastri")
x.setBirthday("10 April 1983")
```

   Run your program and observe that it terminates prematurely, and an error message notifying of `TypeError` is output.

5. To avoid premature termination, add an error handling code using the `try...except` block as follows:

```
try:
    x = Person("Eve Polastri")
    x.setBirthday("10 April 1983")
except TypeError as e:
    print("Type error has occurred", e)
```

   The above code will cause Python to execute the statements in the `try` block up to the point at which `TypeError` occurs. Once this happens, the control will be transferred to the first statement of the `except` block skipping the remaining code within the `try` block. Variable `e` will be bound to an instance of `TypeError`, and when passed as argument to `print`, will evaluate to the message with which `TypeError` was raised (as shown above).

Run the above code and observe that now the program does not terminate abnormally, and the `print` statement in the `except` block takes effect.

6. *OPTIONAL* Modify `__lt__()` to test if `other` is of type `Person`, and raise `TypeError` if not. Modify `getAge()` to test if `self.birthday` is not equal `None`, and raise `ValueError` if it is. Test both methods by calling them with erroneous arguments and making sure proper errors are raised.

5 Adding subclasses

5.1 Create a class `Student` (see the template overleaf), and make it a subclass of `Person`. Specialise `Student` by adding a new data attribute `degree`. In the constructor, call the constructor of `Person`, and then initialize `self.degree` with the value supplied as argument. Add a getter method to return the value of the `degree` attribute. Override the `__str__()` method of the parent class to produce a student specific string as explained in its docstring.

Save your code in a file named `Student.py` (a template is available from Moodle). Make sure to import the `Person` class into the main scope as shown in the template below. Test your implementation as explained in Exercise 3.2.

```
from datetime import date
from Person import Person

class Student(Person):

    def __init__(self, name, degree):
        """
        name, a string
        degree, a string
        Call __init__ of Person to initialise
        the attributes of the superclass followed
        by setting self's degree to the value
        of the degree argument
        """
        pass

    def getDegree(self):
        """
        Getter method for self.degree
        """
        pass

    def __str__(self):
        """
        Returns self's string representation
        combining the values of self's name, birthday, and
        degree attributes.
        To ensure proper information hiding,
        use getName() and getBirthday() methods of the parent
        class to retrieve the values of the name and
        birthday attributes.
        A possible return value could look like
        '(Student:<self.name>:<self.birthday>:<self.degree>)'
        as in '(Person:John Doe:1999-09-09:Data Science)'.
        """
        pass
```

6 Using class variables

6.1 Add an integer *class variable* `nextStudId` to the `Student` class, and use it to assign students unique identifiers as follows:

(a) Make sure `nextStudId` is placed and initialised within the `Student` class scope but **not** inside the method scopes.

(b) Initialise `nextStudId` to 0.

(c) Modify the `Student` class constructor as follows:

i. Add code to initialise a new attribute `self.studId` with the current value of `Student.nextStudId`.

ii. Increment `Student.nextStudId` to make sure the next student to be created will be assigned a different identifier.

(d) Add a getter method `getStudId()` to retrieve the value of `self.studId`.

(e) Add an `__lt__()` method to compare students based on the numeric values of their `studId` attributes.

Test your implementation by creating several instances of the `Student` class, and validate that their identifiers are unique, and reflect the instance creation order. Compare the created instances pairwise making sure the results are consistent with the identifier order.

6.2 Add more subclasses to the `Student` class in `Student.py`, then add code to their methods replacing the `pass` keywords in the template below (available on Moodle):

```
class UnderGrad(Student):
    """
    A class representing an
    undergraduate student.
    It does not add any new attributes
    to or overrides the existing attributes
    of Student. Do not replace pass with anything.
    """
    pass

class PostGrad(Student):
    """
    A class representing a postgraduate student.
    Specialises Student by
```

```
        adding a new data attribute thesis_topic
        along with associated getter and setter methods.
        """

        def __init__(self, degree, name):
            """
            Call the constructor of Student, then
            set self's thesis_topic attribute
            to None
            """
            pass

        def setThesisTopic(self, topic):
            """
            Setter method for self's thesis_topic
            attribute.
            """
            pass

        def getThesisTopic(self):
            """
            Getter method for self's
            thesis_topic attribute
            """
            pass

    class ExchangeStudent(Student):
        """
        A class representing an exchange student.
        Specialises Student by
        adding a new data attribute home_school
        and an associated getter method.
        """

        def __init__(self, name, degree, home_school):
            """
            Call the constructor of Student,
            then set self's home_school
            to the value passed in the
            home_school argument
```

```
        """
        pass

    def getHomeSchool(self):
        """
        Getter method for self's home_school
        attribute
        """
        pass
```

Note that the `UnderGrad` class does not add any new attributes to or overrides any existing attributes of `Student`. The point of having this class is that now, given an instance of `Student`, you can use either `isinstance()` or `type()` to determine to which of the above three categories of students the instance belongs to (see the next exercise).

Add some code in the testing section to create instances of the above classes and test their methods.

6.3 Write a function `print_student_info()` that accepts an instance `person` of `Person`, and outputs the following information about `person`: If the person is a student, `print_student_info()` prints their identifier, followed by their name, followed by "is an ¡undergrad/postgrad/exchange student¿ studying", followed by the degree of study. In addition, if the person is a postgrad, it also prints their thesis topic provided the student has chosen one. If `person` is not a student, but is an instance of `Person`, `print_student_info()` prints their name followed by "is ¡age¿ years old and is not a student". Otherwise, `print_student_info()` raises `TypeError`. Below are a few examples of outputs that will be produced by `print_student_info()` when called with various objects:

```
ug = UnderGrad("Carl Smart", "Computer Science")
pg = PostGrad("Mary Clever", "Information Security")
pg.setThesisTopic("Secure Deep Learning")
ex = ExchangeStudent("Will Homesick", "Music", "Ecole Normale")
tom = Person("Tom Cruise")
tom.setBirthday(date(1962, 7, 3))
```

```
# 1 Carl Smart is an undergrad studying Computer Science
print_student_info(ug)

# 2 Mary Clever is a postgrad studying Information Security with th
# 'Secure Deep Learning'
print_student_info(pg)

# 3 Will Homesick is an exchange student from Ecole Normale studyin
print_student_info(ex)

# Tom Cruise is 56 years old and is not a student
print_student_info(tom)

# TypeError
print_student_info("Sheldon Cooper")
```

`print_student_info()` is an example of a *polymorphic* function, i.e., it is able to process arguments of any type, and produce meaningful results as long as its argument is a subclass of `Person`.