Week 2: Python Worksheet 2, Classes

Barry Rowlingson

November 21, 2021

1 Intro

For these exercises you should be writing Python files and running them from a command line.

2 Animals

Here's the two simple class definition structures for `Animal` and `Dog` as seen in the lectures. This should be `animal.py` in the download folder with these workshop notes.

```python
class Animal():
    noise = ""
    def __init__(self, owner, name):
        self.owner = owner
        self.name = name
    def make_noise(self):
        """ make the noise """
        return self.noise

class Dog(Animal):
    noise = "woof"
    def __init__(self, owner, name, breed):
        super().__init__(owner, name)
        self.breed = breed

d = Dog(owner="Lee Duncan",name="Rin Tin Tin",
        breed="German Shepherd")
print(d)
print(d.make_noise())
```

```
<__main__.Dog object at 0x7f72e5144670>
woof
```

Make sure you can run this from a command line (eg `python animal.py`), and edit it from an editor. When run you should see an output like above - it prints a `Dog` object and then the noise a dog makes.

2.1 Representation

Python uses special methods with names bounded by double underscores for certain tasks related to the function of classes. You've already seen `__init__` which is used to create classes. These are often called "dunder" methods (short for double-underscore).

The `__repr__` method is used to construct the *text representation* of the object when printed. The default method produces text like `<__main__.Dog at 0x7f72e5144670>` as seen above which isn't very informative. Define a `__repr__` method for the `Animal` class to include the animal's name in the representation. It takes just the `self` argument which will be the object itself, as with most methods. Because we define this as a method for `Animal`, it will apply to all classes that are sub-classes of `Animal`, such as `Dog`.

You can use `type(self)` in this method to get the class of the object – in this case it will be the `Dog` class. The name of a class as a text string can be obtained by getting the `__name__` attribute of the class object. In this way you can make the `__repr__` function include the class in the printed output by getting `type(self).__name__`

You can construct strings in Python using the plus sign, or using "format strings" which evaluate expressions in curly brackets. For example:

```
s1 = "Dog"
s2 = "Rin-Tin-Tin"
print(s1 + ":" + s2)
print( f"{s1}:{s2}" ) # f prefixes a format string
```

```
Dog:Rin-Tin-Tin
Dog:Rin-Tin-Tin
```

Use these so that an animal prints with the class name and the animal's name.

```
d = Dog(owner="Lee Duncan",name="Rin Tin Tin",
        breed="German Shepherd")
print(d)
```

```
< Dog: Rin Tin Tin >
```

2.2 More Noise

Rewrite the `make_noise` method to include an extra argument, `n`, so that the noise is repeated `n` times.

```
d = Dog(owner="Lee Duncan",name="Rin Tin Tin",
        breed="German Shepherd")
print(d)
print(d.make_noise(3))
```

```
< Dog: Rin Tin Tin >
woofwoofwoof
```

Hint: see what `"this" * 10` does in Python.

2.3 Louder!

Add an extra argument to the `make_noise` method to control the volume (loudness) of the noise. If called with `volume=0` the animal should be silent, with `volume=1` the normal noise should be returned, and if `volume=2` turn the normal noise into an ALL UPPERCASE string. Hint: there are methods for character strings that can do conversions etc. Try `help('')` to see what methods are available.

Again notice that since this is a method for `Animal` it will automatically apply to any sub-class of `Animal` - all our animals can be loud!

```
d = Dog(owner="Lee Duncan",name="Rin Tin Tin",
        breed="German Shepherd")
print(d.make_noise(2))
print(d.make_noise(3, volume=0))
print(d.make_noise(4, volume=2))
```

```
woofwoof

WOOFWOOFWOOFWOOF
```

2.4 Cat

Create a new class for cats.

```
c = Cat("Sabrina","Salem","Moggie")
print(c)
print(c.make_noise(3, volume=2))
```

```
< Cat: Salem >
MEOWMEOWMEOW
```

3 Student Records

For this section we're going to build a system for storing student exam records.

This will require two new classes, `Mark` for storing course codes and exam scores, and `Student` for the student data.

Write the `Mark` class. You need an `__init__` method that takes a course code and an exam score and stores them as properties of `self`. You can add basic verification of the exam scores here.

The *grade* for a course is defined as follows: 90 or above: "Distinction"; 80 to 89: "Merit", 60 to 79: "Pass" and below 60: "Fail". Note this is *NOT* the official university grade scheme!

Write a method that returns the grade as a string according to this scheme.

Write a `__repr__` method that returns a string with the course code, mark achieved, and the grade.

You should now be able to do something like this:

```
M = Mark("CHIC100",85)
print(M)
print(M.score)
print(M.code)
print(M.grade())
```

```
Course: CHIC100, Score: 85 (Merit)
85
CHIC100
Merit
```

Check that scores produce the right grades:

```
for i in range(101):
    print(Mark("CHICxxx",i))
```

Now write a `Student` class that will represent a student and store their marks. The `__init__` method should be passed the student name and initialise an empty dictionary to hold the course marks. Write a `__repr__` method that prints the student name.

```
S = Student("Fred")
print(S)
print(S.courses) # empty dictionary
```

```
Student Fred
{}
```

Next write a method to add a course mark to the dictionary of courses. Use the course code as the key so that the stored course marks will always be unique in the course code (dictionaries cannot have duplicate keys).

```
C100 = Mark("CHIC100", 84)
C200 = Mark("CHIC200", 74)
S.add_course(C100)
S.add_course(C200)
print(S.courses) # two courses
C200 = Mark("CHIC200",47)
print("Still only two courses:")
print(S.courses) # two courses
```

```
{'CHIC100': Course: CHIC100, Score: 84 (Merit), 'CHIC200': Course:
CHIC200, Score: 74 (Pass)}
Still only two courses:
{'CHIC100': Course: CHIC100, Score: 84 (Merit), 'CHIC200': Course:
CHIC200, Score: 74 (Pass)}
```

To decide the final award for the student the following scheme (not an official University scheme!) is applied:

- If the student has taken fewer than 10 courses: Fail
- If the student has failed 3 or more courses: Fail

- If the student has more than 5 Distinctions: Distinction
- If the student has more than 5 Merits and Distinctions: Merit (unless qualifying for Distinction)
- Otherwise: Pass

Write a method that returns all the grades for the students courses by iterating over the values of the dictionary holding the courses and getting the grade for the course.

```
S.all_grades()
```

```
['Merit', 'Pass']
```

Write a method that uses the `all_grades` method, and counts all the occurrences of the four possible classifications: Fail, Pass, Merit and Distinction. Have it return a dictionary with the classification as key and the count as value.

```
S.grade_summary()
```

```
{'Total': 2, 'Fail': 0, 'Pass': 1, 'Merit': 1, 'Distinction': 0}
```

Finally write an `award` method that calls the `grade_summary` method and returns the award according to the scheme listed above.

```
S.award()
```

```
'Fail, only 2 courses'
```

Finally you should be able to put something like this `__main__` block into your module file so you can run the file as a test:

```
if __name__ == "__main__":
    M = Mark("CHIC100",84)
    print(M)

    J = Student("Joe")
    J.add_course(M)

    print(J," is awarded ",J.award())

    S = Student("Sam")
    ## add 10 courses to Sam, with marks from 85 to 95
    for i in range(10):
      C = Mark("CHIC"+str(100+i), 85+i)
      S.add_course(C)
    print(S, " is awarded ",S.award())
    print(S.grade_summary())
```

```
Course: CHIC100, Score: 84 (Merit)
Student Joe  is awarded  Fail, only 1 courses
Student Sam  is awarded  Merit
{'Total': 10, 'Fail': 0, 'Pass': 0, 'Merit': 5, 'Distinction': 5}
```

There are a number of possible enhancements to this code:

- Using text strings for the course grading ("Pass", "Fail", etc) is a bit "fragile" since it can be easily broken by a typo or a change in definition. We should create a new class and define objects for each grade.
- The code for computing the grade for a course is buried in the `Mark` class. This makes it hard to quickly test the code, since we have to build a `Mark` object, then see what grade is returned. Ideally we'd have a standalone function that took a number and returned a grade, and then the method would call this.
- Similarly, the code for computing the degree award is buried in the `Student` class. This makes it hard to test since we have to create a `Student` object, add courses to it, and then see what award is returned. This should really be in a standalone function that takes a list of grades and returns the corresponding award.

Never be afraid to revise version 1 of your code.