Practical 7 - Classes & Objects

Did you finish last week's work? If not, make sure to complete it during the week. If you do not understand anything, bring those questions to your tutor the following week.

We have seen how to work with lists, tuples and dictionaries to store and process data appropriate for those types:

- **list** is useful for storing an ordered sequence of data (e.g. monthly rainfall data)
- **tuple** is useful for storing fixed (not changing) data with multiple parts (e.g. date of birth)
- **dict** is useful when the data has a 'mapping' relationship (e.g. names -> dates of birth)

Very often we want to combine data into one object in a way that doesn't suit one of the built-in types, so we write our own **classes** for these situations. That's what this practical is all about. As you do the prac, pay attention to how the class is helping us combine data and functions in one entity. Ask questions as needed!

Walkthrough Example

Get car.py from (remember to click Raw) https://github.com/CP1404/Practicals/blob/master/prac_07/car.py and add it to your PyCharm project in this week's prac folder.
Create a new file called "used_cars.py" and add the following code, or copy (Raw version) https://github.com/CP1404/Practicals/blob/master/prac_07/used_cars.py

```
from prac_07.car import Car

def main():
    my_car = Car(180)
    my_car.drive(30)
    print("fuel =", my_car.fuel)
    print("odo =", my_car.odometer)
    print(my_car)

main()
```

Note that the import statement assumes you have your car.py file in a folder called prac_07 as we suggested.
Run your program and it should work.
Spend some time studying the Car class.

Things to do:

In the used_cars program file, write one new line of code for each of the following:

1. Create a new Car object called "limo" that is initialised with 100 units of fuel.
2. Add 20 more units of fuel to the car using the add method.
3. Print the amount of fuel in the car.
4. Attempt to drive the car 115km using the drive method.
5. Print the car's odometer reading.

6. Now add the `__str__` method to the Car class in car.py.
   Using {} string formatting, have it return a string in the following format:
   `Car, fuel=42, odometer=277`
   Remember that you can run this method by **print**ing your car object, or passing the car object to the **str()** function. Do NOT call the method explicitly like `my_car.__str__()`

7. Now add a "**name**" field to the Car class (in car.py), and adjust the `__init__` and `__str__` methods to set and display this respectively. Make the str method return the car's name instead of just "Car".
   Now **add names** to the constructors where you create Car objects in the used_cars.py program.

Intermediate Exercises

Let's make our own simple class for a **programming language**.
Create a new file for our class - `programming_language.py`
Call your class `ProgrammingLanguage` (using Python's recommended "PascalCase" or "CapWords" style)

There are lots of things we could store, but we'll consider only a few, based mostly on the information found at this Programming Language Comparison page.
For each language, we want to store the following fields - the row names from this table:

(Field)	**Java**	**C++**	**Python**	**Visual Basic**	**Ruby**
Typing	Static	Static	Dynamic	Static	Dynamic
Reflection	Yes	No	Yes	No	Yes
Year	1995	1983	1991	1991	1995

Define the following **methods**:

- **`__init__`** - like most init functions, create the fields and set them to the parameters passed in
- **`is_dynamic()`** - which returns True/False if the programming language is dynamically typed or not
  Note: it's *really important* that you understand this function will take no parameters (other than self). The information is already stored *inside* the object, so you don't need to tell the object its data. This function's purpose is to encapsulate the Boolean functionality that would make the class more helpful. See how the function name starts with "**is**", like isupper(), isnumeric(), etc.? So, it returns a Boolean.

Create a simple program, `languages.py`.
Import the class, then copy these 3 languages:

```
ruby = ProgrammingLanguage("Ruby", "Dynamic", True, 1995)
python = ProgrammingLanguage("Python", "Dynamic", True, 1991)
visual_basic = ProgrammingLanguage("Visual Basic", "Static", False, 1991)
```

Now add the **`__str__`** method, which should return a string like:

```
Python, Dynamic Typing, Reflection=True, First appeared in 1991
```

Print the python object and see if your `__str__` function is working properly.

Now create a new list that contains these three existing ProgrammingLanguage objects.

Do this next part on paper first, then copy it into PyCharm to see how you went. Remember that writing code on paper (or a whiteboard) is good practice, helps you learn it better (since you can't depend on the IDE's help) and encourages you to be consistent and clear with syntax, indenting, etc.

Loop through and print the names of all of the languages with dynamic typing, which should produce:

```
The dynamically typed languages are:
Ruby
Python
```

Do-from-scratch Exercises

Guitars!

Remember the string formatting example from prac 2:

```
name = "Gibson L-5 CES"
year = 1922
cost = 16035.40
print("My guitar: {0}, first made in {1}".format(name, year))
```

You should notice that we have multiple values to store for one guitar entity: name, year and cost... and that guitars are awesome!
What if we owned 9 guitars? We'd want to use a collection like a list… but what would each element in the list be? … A tuple? A dictionary? No… This is a classic case for a class!

Write a **Guitar** class that allows you to store one guitar with those **fields** (attributes):

- **name** (we could split this into make and model, but one name field will do us for now)
- **year**
- **cost**

Define the following **methods**:

- **`__init__`** - with defaults name="", year=0, cost=0
- **`__str__`** - which uses {} string formatting to return something like (using the values from above):
  `Gibson L-5 CES (1922) : $16,035.40`
- **`get_age()`** - which returns how old the guitar is in years (e.g. the L-5 is 2017 - 1922 = 95)
- **`is_vintage()`** - which returns True if the guitar is 50 or more years old, False otherwise
  Hint: try using get_age() to simplify the implementation of this method!

Remember that methods should not take in any data that the object already knows (like age, year, etc.).

Testing

Now **write a `guitar_test.py` program** with at least enough code to test that the last two methods work as expected.
So to test that the **get_age()** method works, you could test that the above example guitar does indeed output 95 as expected. Here is some sample output for testing two guitars where the second called `Another Guitar` has year=2012):

```
Gibson L-5 CES get_age() - Expected 95. Got 95
Another Guitar get_age() - Expected 5. Got 5

Gibson L-5 CES is_vintage() - Expected True. Got True
Another Guitar is_vintage() - Expected False. Got False
```

Do you see how this works?
We print our own *literal* for what we expect if it's right (e.g. 95), then we print *what the actual method returns* and we look at the output to see if they match. This form of testing is quite 'manual' since we need to read the output and compare it ourselves, but it is a good start.

Let's say we wrote the is_vintage() method incorrectly, then we want to see something like:

```
50-year old guitar is_vintage() - Expected True. Got False
```

We can see that the actual does not match the expected, so we know we need to fix something.

Playing the Guitars (not really)

Great! Now that you have the class tested a bit, write a program that uses it: `guitars.py`
The program should use a list to store all of the user's guitars (keep inputting until they enter a blank name), then print their details.
Read the full question including the notes before starting. We've written helpful comments to make it easier and to teach you useful lessons.

Sample Output (`green` is user entry):

```
My guitars!
Name: Fender Stratocaster
Year: 2014
Cost: $765.4
Fender Stratocaster (2014) : $765.40 added.
Name:
      … snip …
These are my guitars:
Guitar 1:  Fender Stratocaster (2014), worth $    765.40
Guitar 2:       Gibson L-5 CES (1922), worth $ 16,035.40 (vintage)
Guitar 3:        Line 6 JTV-59 (2010), worth $  1,512.90
```

Programmer Efficiency Note:

When testing a program like this you can waste a lot of time typing in input… then changing something, running it again and… typing the same thing again…
So don't do it!

Instead, comment out the user input lines, and put in lines like this to 'get' the data for testing:

```
guitars.append(Guitar("Gibson L-5 CES", 1922, 16035.40))
guitars.append(Guitar("Line 6 JTV-59", 2010, 1512.9))
```

According to Wikipedia's page on the abstraction principle, "When read as recommendation to the programmer, the abstraction principle can be generalised as the "*don't repeat yourself*" principle, which recommends avoiding the duplication of information in general, and also avoiding the duplication of human effort involved in the software development process."

Notes:

- The sample output uses some nice string formatting. Feel free to try and figure this out, or just use our code (the width we use for `guitar.name` is just a guesstimate):

  ```
  print("Guitar {}: {:>20} ({}), worth ${:10,.2f} {}".format(i + 1,
  guitar.name, guitar.year, guitar.cost, vintage_string))
  ```

  The variable `vintage_string` is set to "" or "(vintage)" depending on the `is_vintage()` method.
  Note: If you're keen, try using Python's *ternary operator* (search for it) to do this in one line.

- See guitar.year, guitar.cost...? You can do this another way if you want...
  E.g. for the car class example above, the following lines are equivalent. This is sometimes a useful way to make the code more readable because you can see the name of the variable you're printing in the actual placeholder.

  ```
  print("Car {}, {}".format(my_car.fuel, my_car.odometer))
  print("Car {self.fuel}, {self.odometer}".format(self=my_car))
  ```

- For this particular code, we've used both `i` and the target variable `guitar` (instead of `guitars[i]`) by using the built-in **`enumerate()`** function. You don't have to do it this way, but if you want to, it's like this:

  ```
  for i, guitar in enumerate(guitars):
  ```

  … So `enumerate()` must return what type? A tuple!

Practice & Extension Work

Use these exercises as much-needed practice and as ways to learn new things.

1. Using car.py, write a **car driving simulator**, with output like the following…
Note: Please do this (and every problem of significant size) incrementally:
- Start by just testing one method call,
- then another,
- then write the menu and put it all together.
  (Do not start with the menu.)

Remember to use the class's functionality - don't rewrite anything you've already got.

Do you remember how to construct a simple menu-driven program? If not, it's very important that you revise earlier lectures and practicals.
You will need to `import` the car module, create a Car object, and use appropriate methods on that object.

```
Let's drive!
Enter your car name: The bomb
The bomb, fuel=100, odo=0
Menu:
d) drive
r) refuel
q) quit
Enter your choice: f
Invalid choice

The bomb, fuel=100, odo=0
Menu:
d) drive
r) refuel
q) quit
Enter your choice: d
How many km do you wish to drive? 39
The car drove 39km.

The bomb, fuel=61, odo=39
Menu:
d) drive
r) refuel
q) quit
Enter your choice: d
How many km do you wish to drive? -25
Distance must be >= 0
How many km do you wish to drive? 100
The car drove 61km and ran out of fuel.

The bomb, fuel=0, odo=100
Menu:
d) drive
r) refuel
q) quit
```

```
Enter your choice: r
How many units of fuel do you want to add to the car? -80
Fuel amount must be >= 0
How many units of fuel do you want to add to the car? 120
Added 120 units of fuel.

The bomb, fuel=120, odo=100
Menu:
d) drive
r) refuel
q) quit
Enter your choice: d
How many km do you wish to drive? 25
The car drove 25km.

The bomb, fuel=95, odo=125
Menu:
d) drive
r) refuel
q) quit
Enter your choice: q

Good bye The bomb's driver.
```

2. Create a **Date** class
Store the fields:

- **day**
- **month**
- **year**

Write some useful methods, including:

- **init and str**
- **add_days(n)** - which should add n days to the stored date (perhaps harder than it seems)

Test the class.
Note: Python has built in date and time functionality in the **datetime** module, so you wouldn't usually write your own class to store a date, but this is a good practice exercise.

3. Create a program that uses a list of **Person** objects. Each Person object records the first-name and last-name of each person along with their age. The user can type in the details of any number of people. The code generates a table formatted with the first-names, last-names, and ages of the people (perhaps sort the people into order based on their ages).

You should try using the **command line** for Git soon. It's a valuable skill. On the lab computers, you should be able to use "Git Bash". Right-click on the folder where your files are and select "Git Bash here". On a Mac, just use Terminal.

A great place to learn git commands is: https://try.github.io

Why? Here's a quote from one of our students who completed this subject:
"Now I'm moving into 2nd yr subjects, I found the integrated git/pycharm a good intro step, but I'm finding git bash with a simple cheat sheet is a nice progression. I think 99% of us could use some more training/practice with branching, merging, etc. Still kinda confusing in the terminology."

The more you have to do by yourself, the more you will really understand what's happening. Some things (like removing a file from the index) cannot be done with the integration. So, as soon as you are happy to, start getting used to using git from a console.