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.

Last time, we saw 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 data with multiple parts (e.g. date of birth)
* **dict** is useful when the data has a ‘mapping’ relationship (e.g. names -> ages)

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.

# Walkthrough Example

Download car.py from <https://github.com/CP1404/Practicals2016/blob/master/Prac06/car.py> and add it to your PyCharm project in this week's prac folder.

Create a new file called “usedCars.py” and add this code, or copy <https://github.com/CP1404/Practicals2016/blob/master/Prac06/usedCars.py>

**from** Prac06.car **import** Car

**def** main():

bus = Car(180)

bus.drive(30)

print(**"fuel ="**, bus.fuel)

print(**"odo ="**, bus.odometer)

print(bus)

main()

Note that the import statement assumes you have your car.py file in a folder called Prac06 as we suggested.

Run your program and it should work.

Spend some time studying the Car class.

## Things to do:

In the usedCars program file, write a line of code for each of the following:

1. Create a new Car object called “limo” with 100 units of fuel.
2. Add 20 more units of fuel to the car.
3. Display the amount of fuel in the car.
4. Attempt to drive the car 115km.
5. Display 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.
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 so the str method prints the car’s name instead of “Car”.

# Intermediate Exercises

Let’s make our own simple class for a **programming language**.

Create a new file for our class - programminglanguage.py

Call your class ProgrammingLanguage (which uses Python’s recommended “PascalCase” 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](http://www.jvoegele.com/software/langcomp.html) page.

For each language, we want to store the following fields - the row names from this table:

|  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- |
|  | **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\_\_** - create the fields and set them to parameters passed in
* **is\_dynamic()** - which returns True/False if the programming language is dynamically typed or not

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)

vb = 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 put your existing ProgrammingLanguage objects into a list.



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 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 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 2016-1922 = 94)
* **is\_vintage()** - which returns True if the guitar is 50 or more years old  
  Hint: try using get\_age() to simplify the implementation of this method!

### Testing

Now write a guitarTest.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 93. Here is some sample output for testing two guitars (the second has year=2011):

get\_age() - Expected 94. Got 94

get\_age() - Expected 5. Got 5

is\_vintage() - Expected True. Got True

is\_vintage() - Expected False. Got False

### 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.

#### 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](https://en.wikipedia.org/wiki/Abstraction_principle_(programming)), “When read as recommendation to the programmer, the abstraction principle can be generalized as the "[*don't repeat yourself*](https://en.wikipedia.org/wiki/Don%27t_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.”

**Note:**

* 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.

**Another (side) note:**

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 in that you can see the name of the variable you're printing in the actual placeholder.

print(**"Car {}, {}"**.format(bus.fuel, bus.odometer))  
print(**"Car {self.fuel}, {self.odometer}"**.format(self=bus))

* 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.)

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. You wouldn't usually write your own class to store a date.

3. Create 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/GitHub 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”.  
A great place to learn this is:

<https://try.github.io>