

# GCIS-123

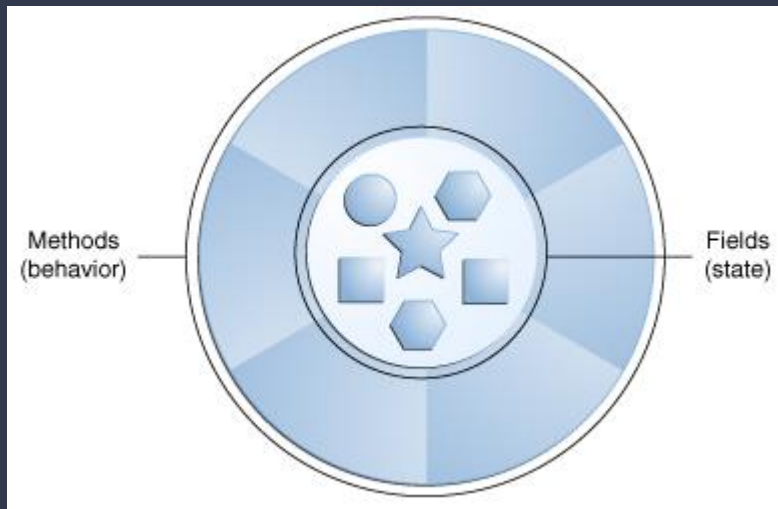
## Software Development & Problem Solving

*Python Methods*

**RIT**

Golisano College of  
Computing and  
Information Sciences

# Methods



This unit, we will continue the exploration of **classes** and **encapsulation** by introducing **methods**.

Data stored in **fields** define the **state** of a class. We will use **methods** to define the **behavior**.

- In the last unit we explored the object-oriented concept of **encapsulation**.
  - A class defines the **state** that belongs to a category of things.
  - Each object acts like a container that holds together its copy of the state.
  - We also refer to this state as **fields**.
- In this unit we will continue to explore **encapsulation**.
  - A class may also define the **behavior** of a category of things.
  - We refer to this behavior as **methods**.
- We will explore:
  - Methods
  - Protected Access
  - Special Methods
- To start, we will specifically focus on:
  - Adding functions to classes.
  - Protecting access to variables in an object.

# Review: Classes & Objects

- A **class** is a **blueprint** or **template** that defines the characteristic of some category of things.
  - All members of this category have these characteristics.
  - For example, all Students have an id, name, credits, and gpa.
- The class is used to **construct instances**.
  - An instance is also sometimes called an **object**.
  - Each **instance** has its own copy of the attributes defined by its class.
  - For example, changing the name of one student does not affect the name of any other students.
- In this way, an instance acts like a **container** that holds its own copies of the **fields** defined by its class.
  - This is called **encapsulation**.
- Python allows programmers to **dynamically** attach fields to objects at runtime.
  - This is risky and can cause lots of problems and confusion.
  - Two instances of the same class may have different fields!

**Static fields** can be declared inside a class. A **shallow copy** is made for each **instance**. This can be problematic with **reference types** like lists.

```
1 class Bicycle:
2     color = "black"
3     gears = 21
4     training_wheels = False
5
6 bike = Bicycle()
7 bike.gears = 1
8 bike.training_wheels = True
9 bike.streamers = True
10 print(bike.training_wheels)
```

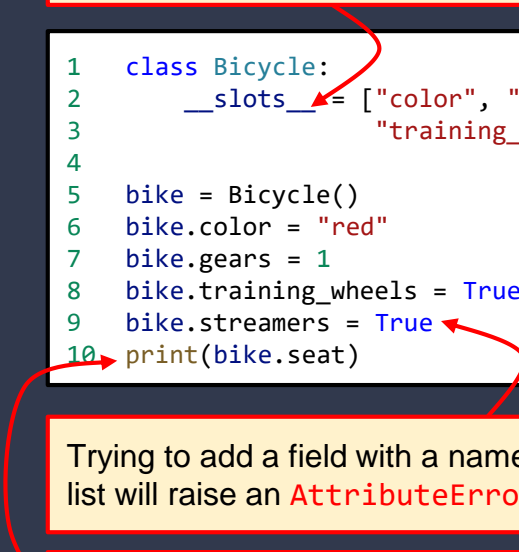
While each instance gets a copy of each field with the default value, **dot-notation** can be used to **change** the values.

Python also allows new fields to be added to instances **dynamically** using **dot-notation**.

# Review: `__slots__`

The specially named `__slots__` **static field** can be used to explicitly define the list of acceptable field names in a class.

```
1 class Bicycle:
2     __slots__ = ["color", "gears", "seat",
3                 "training_wheels"]
4
5 bike = Bicycle()
6 bike.color = "red"
7 bike.gears = 1
8 bike.training_wheels = True
9 bike.streamers = True
10 print(bike.seat)
```



Trying to add a field with a name that is **not** in the list will raise an **AttributeError**.

But just because a name is in `__slots__` doesn't mean that it's added to the instance! Trying to use a field that hasn't been attached will raise an **AttributeError**, even if its name is in `__slots__`.

- Python allows programmers to **attach** fields **dynamically** to instances of a class.
- This is very powerful and flexible!
- But it also leads to lots of potential problems.
  - Different instances of a class may have different fields, which can be confusing.
  - **Misspellings** and **typos** can lead to problems like the "flippable.crad" vs. "flippable.card" example.
- Python does provide a mechanism for restricting the names of possible fields to a specified list: the `__slots__` static field.
  - This specially named static field is used to explicitly articulate the names of any fields that **may** be added to an instance of the class.
  - Defining a `__slots__` field **does not** automatically add fields to the class! They must still be added programmatically in some way.
  - However, each time a new field is added to an instance of the class, it is checked against the list of names in the `__slots__` list.
  - If the name is not present in `__slots__`, an **AttributeError** is raised.
- You cannot have **static fields** and `__slots__` with the same name!

## 11.2 A Pet Class

Object-oriented programming is an entirely new way of thinking! We only just learned about classes and objects in the last unit, so let's get a little more practice. Begin implementing a class that represents a pet.



- Create a new Python module in a file named "pets.py" and begin implementing a `class` to represent a `Pet`.
  - Your `Pet` class should use the special variable name `__slots__` to define a list of fields including `species`, `name`, `weight`, `fur_color` and `age`.

```
class Bicycle:
    __slots__ = ["color", "gears", "seat",
                 "training_wheels"]
```

- Using `__slots__` can prevent programmers from making careless errors and adding the wrong fields to a class, but constructing classes in this way is still a bit *clunky*.
  - Construct an empty instance.
  - Add fields to it one at a time.
  - Repeat every time you make an instance.
- A *much* better way to initialize a class is to add a **constructor**.
  - A function declared **inside** the class that is named `__init__` and declares at least one parameter: `self`
  - `self` is a reference to the object that is being constructed.
  - Fields can be added to the object using **dot-notation**, e.g. `self.color = "red"`
- Additional parameters may be added to the constructor as well.
  - These parameters can be used to **initialize fields**.
  - Other fields may be assigned **default values** or values that are **derived** from the parameters.
  - Be careful with **reference types**, e.g. lists, as default values. As with static class variables, all instances receive a reference to a **shallow copy**. If you must have a reference type as a default value, make a **copy** before assigning it to your **object attribute**.

# Review: Constructors

A **constructor** uses the special name `__init__` and must declare a parameter for `self`, which is a reference to the object being constructed.

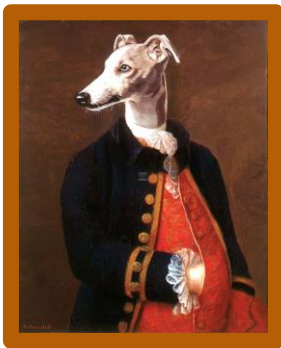
```
1 class Bicycle:
2     __slots__ = ["color", "gears", "seat",
3                 "training_wheels"]
4
5     def __init__(self, color, seat):
6         self.color = color
7         self.seat = seat
8         self.gears = 21
9         self.training_wheels = False
10
11 bike = Bicycle("red", "banana")
11 print(bike.training_wheels)
```

The constructor may declare additional parameters as well. When invoked, arguments **must** be provided for all parameters **except** for `self`.

Using a constructor provides a much cleaner, simpler way to construct a class **and** guarantees that all of the fields are initialized.

## 11.3 A Pet Constructor

Using `__slots__` with a constructor can help to ensure that every instance of your class has the same fields as any other instance. Let's add a constructor to the `Pet` class that initializes all of the fields using parameters and/or default values.



```
class Bicycle:
    __slots__ = ["color", "gears"]

    def __init__(self, color):
        self.color = color
        self.gears = 21

bike = Bicycle("red")
```

- Open the `pets` module and add a constructor to the `Pet` class.
  - The method should use the special name `__init__`
  - You should declare a parameter for `self` so that the object being constructed can **refer to itself**.
  - Declare parameters for each of the fields defined in `__slots__`.
    - The `age` parameter should have a default value of `0`.
  - Set the fields on the new `Pet` using the constructor parameters.
- Create a new Python module in a file named `"pets_main.py"` and add a `main` function that uses the constructor to create at least one instance of your `Pet` class.
  - Don't forget to `import pets`.
  - Print its `name` and `weight`.

# Methods

- Up to this point we have exclusively focused on the **state** that classes **encapsulate**.
  - The fields that are contained by objects of the class.
- But classes may also encapsulate **behavior**.
  - Functions can be added to the class.
  - Every function declares at least one parameter that refers to the object on which the function is being called: **self**.
  - The functions may also declare other parameters as needed.
- A function that belongs to an object is called a **method**.
  - A method **should** interact with the fields in the object in some way, e.g. by changing or using them.
  - Otherwise, why is the method in the class?

A **method** is declared inside of a class and the first parameter is always **self** - a reference to the object on which the method is being called.

```
1 class Bicycle:
2     # __slots__ and constructor not shown
3
4     def ride(self, name):
5         print(name, "rides their",
6               self.color, "bicycle with",
7               self.gears, "gears and a",
8               self.seat, "seat!")
9
10        if self.streamers:
11            print("With streamers!")
```

A **method** should **interact** with the fields in an object in some way. Otherwise, it may not need to be part of the class.



## 11.4 Feeding Your Pet

Methods are functions that are declared inside of a class. The method is called on an object of the class using dot notation. Let's practice by adding a method to the `Pet` class that feeds it a specified number of calories.



```
class Bicycle:
    # __slots__ and constructor not shown

    def ride(self, name):
        print(name, "rides their",
              self.color, "bicycle with",
              self.gears, "gears and a",
              self.seat, "seat!")
```

- Open the `pets` module and add a method to the `Pet` class named "`feed`" that declares a parameter for an amount of `calories`.
  - Don't forget that the first parameter to any method must be `self` so that the object on which the method is called can **refer to itself**.
  - The pet should gain `weight` according to the formula:
    - `pounds = calories / 3000`.
- In your `pets_main` module, update the `main` function to print the pet's `name` and `weight` before and after feeding it `1800 calories`.
  - The weight should increase by `0.6` pounds.

## 11.5 Walking Your Pet

Classes may have many more than one method. Let's add another method to the Pet class that walks it a specified distance. Pets that walk burn calories and lose weight.



```
class Bicycle:
    # __slots__ and constructor not shown

    def ride(self, name):
        print(name, "rides their",
              self.color, "bicycle with",
              self.gears, "gears and a",
              self.seat, "seat!")
```

- Open your pets module and add a method to your Pet class named "walk" that declares a parameter for a distance in miles.
  - Don't forget that the first parameter to any method must be `self` so that the object on which the method is called can refer to itself.
  - The pet should lose weight according to the formula:
    - `pounds = miles * 100 / 3000`.
  - In your `pets_main` module, update the `main` function to print your pet's name and weight before and after walking it 1.5 miles.
    - The weight should decrease by 0.05 pounds.

# More Encapsulation

## (Private Fields)

In Python, naming a field with a double-underscore marks it as **private**. The field names in `__slots__` should include the prefix.

```
1 class Bicycle:
2     __slots__ = ["__color", "__gears",
3                 "__seat", "__training_wheels"]
4
5     def __init__(self, color, seat):
6         self.__color = color
7         self.__gears = 21
8         self.__seat = seat
9         self.__training_wheels = False
10
11 bike = Bicycle("red", "banana")
12 print(bike.__gears)
```

The double underscore should be used whenever referring to a private field from inside the class.

Trying to **directly access** or change the field from outside of the class raises an **AttributeError**.

- Our Pet class currently has five fields representing its **species**, **name**, **weight**, **fur\_color**, and **age**.
- All of these fields are currently **visible** outside of the class.
  - Meaning that any other part of the program can **access** or **change** them.
- But does it make sense for **any** part of the program to be able to change **any** of these values?
  - `dog.age = -10`
  - `cat.species = "buffalo"`
  - `rat.fur_color = "fluorescent purple"`
- An object should **protect** and **control** access to its fields so that other parts of the program can't make arbitrary and nonsensical changes.
- Naming a field with a **double underscore prefix** (`__`) indicates that the field is **private**.
  - e.g. `self.__name = "Michelangelo"`
  - The Python interpreter makes sure that other parts of the program **cannot** directly access or change the field.
  - In general, fields should always be **private**.
- Making fields private is an important aspect of **encapsulation**.
  - This is referred to as **data privacy**.

## 11.6 Pet Data Privacy

Encapsulation (including data privacy) is an important component in object-oriented programming. An object should protect and control access to its private data. Make the fields in your `Pet` class private so that they are protected from outside modification.



```
class Bicycle:
    __slots__ = ["__color", "__gears"]

    def __init__(self, color, seat):
        self.__color = color
        self.__gears = 21
```

- Open your `pets` module and modify all of the fields in your `Pet` class so that they are **private**.
  - Rename all of the fields in your `__slots__` so that they start with a **double underscore** (`__`).
  - Make sure to update references to the fields in your `__init__`, `feed`, and `walk` methods.
- Rerun your `pets_main` module.
  - What happens?
  - What happens when you try to fix it to use the double-underscore?

- Once a field has been made private, it can no longer be **directly** accessed from outside of the class.
  - This is a good thing! An object should control and protect access to its fields!
- But what if other parts of the program need to access a field?
  - e.g. to print it, copy it, etc.
- We add a special method to the class called an **accessor**.
  - The method is named the same as the field with a "get\_" prefix, e.g. "get\_color".
  - The method only declares a **single parameter**: self.
  - The method **returns** the current value of the field.
  - Be careful with reference types such as lists or sets!
  - If you must return an attribute that is a reference type, return a copy, e.g. "return list(self.\_\_my\_list)"
- And what if other parts of the program need to **change** a field?
  - e.g. add streamers to a bike?
- We add a special method called a **mutator**.
  - The method is named the same as the field with a "set\_" prefix, e.g. "set\_streamers".
  - The method declares **two parameters**: self, and a parameter for the new value for the field.
  - The method **assigns** the parameter to the field.
- **Carefully** decide which fields need read and/or write access from outside of the class!

# Accessors & Mutators

**Accessors** only declare a parameter for self and return the current value of the parameter. They are often called "getters."

**Mutators** declare a parameter for self and the field being changed. They are often called "setters."

```
1 class Bicycle:
2     # slots and constructor not shown
3
4     def get_gears(self):
5         return self.__gears
6
7     def set_color(self, color):
8         self.__color = color
9
10 bike = Bicycle("red", "banana")
11 print(bike.get_gears())
12 bike.set_color("pink")
```

Both methods **open encapsulation**, and so should only be written if access is **required** in other parts of the program. Otherwise the fields should be **protected**.

## 11.7 Accessors

Now that we have properly encapsulated all of the `Pet`'s fields by making them private, we can no longer directly access them in other parts of the program! That's a good thing, but it also means that the values can't be used outside of the class at all. Let's add a couple of accessors for the `Pet`'s `name` and `weight` so that other parts of the program can see them.



```
class Bicycle:
    # slots and constructor not shown

    def get_gears(self):
        return self.__gears
```

- Open your `pets` module and add `accessors` for the `Pet`'s `name` and `weight`.
  - Add a method called "`get_name`" that declares a parameter for the object's `self`.
    - Return the pet's `name`.
  - Add a second method called "`get_weight`" that declares a parameter for the object's `self`.
    - Return the pet's current `weight`.
- In your `pets_main` module, update the `main` function so that it **uses the accessors** to print the pet's name and weight.

# Students Revisited



Let's fix our students module to incorporate best practices like private fields, accessors, mutators, and some useful methods!

- All students have the following attributes:
  - A **unique ID**.
  - A **name**.
  - Earned **credits**.
  - Cumulative **GPA**.
- In the last unit we implemented a **Student** class that included:
  - **\_\_slots\_\_** for each of the required fields.
  - A **constructor**.
  - A **separate function** for printing instances of the Student class.
- Now that we have learned about private fields and methods, let's update our Student module to add some new features:
  - Properly encapsulated (**private**) fields.
  - **Accessors** & **mutators** (as needed).
  - A list to keep track of **courses** taken.
  - **Methods**! Including one to compute the student's cumulative GPA.
- Remember that every course has a name, a number of credits, and a grade. We may need a new class...

# 11.8 Encapsulating Students

Let's practice using proper encapsulation in another class. Update the `Student` class so that it uses proper encapsulation for all of its fields. Use your `Pet` class as a reference!



Proper encapsulation (including data privacy) is an important aspect of object oriented programming.

- Open the `students` module and update the `Student` class to use proper encapsulation.
  - Add the **double underscore** (`__`) to the name of each of the fields in your `__slots__`.
  - Make sure to **update the constructor**.
  - Add **accessors** for each field.
  - Which, if any, of the fields should be mutable?
    - Add **mutators** for any such fields.
- Create a new Python module in a file named `"students_main.py"` add add a `main` function.
  - Don't forget to `import students`.
  - Make an **instance** of the `Student` class that represents **you**.
  - Use the **accessors** to **print** the value of the student's fields.



## 11.9 Printing a Student

Encapsulation refers to both *data privacy* and the fact that an object acts like a container that holds its state (fields) and behavior (methods) together in one place. The `print_student` function uses the data inside the `Student` class, but is located *outside* of the class. Let's encapsulate it inside of the class instead.



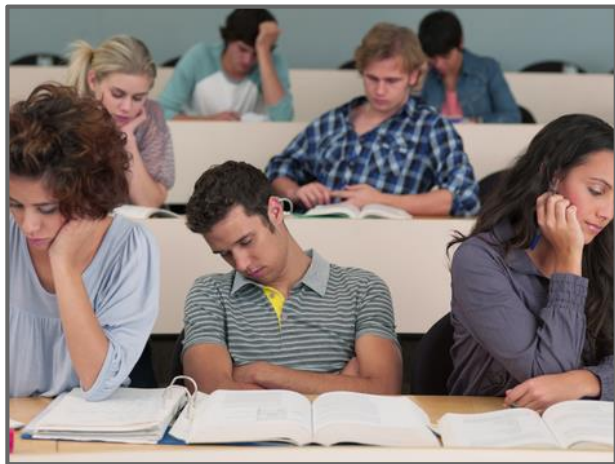
This is a little clunky. We'd like to just pass a student into the built-in `print()` function.

We'll see how to make that work in later in this unit!

- Open your `students` module and navigate to the `print_student` function. It is currently written so that a student is passed in as a parameter and printed. This will no longer work because the fields are all private.
- Besides, a `student` should be responsible for **printing itself**. Encapsulate the function by moving it into the `Student` class.
  - Change the parameter to "`self`" instead of "`student`".
  - Change the print statement to access the newly private fields.
- Change the `main` function in your `students_main` module to use the newly encapsulated `print_student` function to have the student print itself.

## 11.10 A Course Class

Students in our university system need to be able to register to take courses. We'll need to create a class that represents a course and all of its attributes so that we can keep track of the courses that a student may want to take.



Use the Pet and Student classes that you have already written as a reference for your new class.

- Open your `students` module and add a new `Course` class that represents a course taken by a student.
  - It should have fields for `name` (e.g. "GCIS-123"), `credits`, and a `grade`.
    - Use proper **encapsulation**!
  - Write a **constructor**.
  - Write **accessors** for each of the fields.
  - Should any of the fields be **mutable**?
- Why not add a `print_course` method while you're at it?
  - Model it after the `print_student` method.
- Create at least **two instances** of the `Course` class in the `main` function of your `students_main` and use the accessors to print each field.
  - Use courses that you are currently taking!

# Computing GPA

- When computing GPA, each letter grade is worth a different number of **quality points**.
  - A = 4
  - A- = 3.67
  - B+ = 3.33
  - And so on...
- Each course contributes points equal to the quality points multiplied by the credits that the course is worth.
  - For example, if a student receives a C (2 quality points) in a 3-credit course, the course contributes  $2 * 3 = 6$  points to their GPA.
- The overall GPA is the sum of points from all courses divided by the total number of credits.

<u>Course</u>	<u>Grade (QP)</u>	<u>Credits</u>	<u>Points</u>
GCIS-123	B+ (3.33)	4	13.32
SWEN-101	A (4.0)	1	4.0
MATH-181	C- (1.67)	3	5.01
PSYC-101	B (3.0)	3	9.0
PHIL-101	C+ (2.33)	3	6.99
Total		14	38.32
GPA			2.74

## 11.11 Course Points

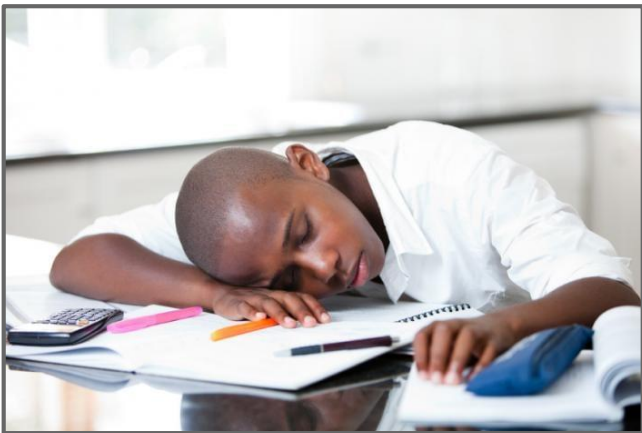
Each `Course` contributes points based on the student's grade (quality points) and the number of credits that the course is worth. Add a method to the `Course` that will compute and return the number of points that it contributes to the GPA.

<u>Course</u>	<u>Grade (QP)</u>	<u>Credits</u>	<u>Points</u>
GCIS-123	B+ (3.33)	4	13.32
SWEN-101	A (4.0)	1	4.0
MATH-181	C- (1.67)	3	5.01
PSYC-101	B (3.0)	3	9.0
PHIL-101	C+ (2.33)	3	6.99
Total		14	38.32
GPA			2.74

- Open the `students` module and take a few moments to examine the `GRADES` list and the `QUALITY_POINTS` dictionary that have been provided to you.
  - You can use the dictionary to quickly retrieve the quality points for any grade.
  - If you have any questions, ask them now!
- Define a new method in the `Course` class named `"get_points"` that computes and returns the points that the course contributes to a student's GPA.
  - Use the course `grade` to get the quality points and multiply it by the course `credits`.

## 11.11 Tracking Courses

Each instance of the `Student` class represents an individual student and each student will need to keep track of their classes. Let's add a private field to the `Student` class to store the courses that the student has taken. What kind of data structure should you use? Make sure to initialize it in the constructor!



- Open your `students` module and add a **new, private field** to the `Student` class to keep track of **courses** taken by the student.
  - Add the name to your `__slots__`. Don't forget the **double underscore** (`__`) to make it private!
  - In the constructor, initialize the field to an empty list.
  - **Do not** add an accessor for the list. Why?

What bad things might happen if another part of the program got access to a student's list of courses?

# 11.12 Computing GPA

Every student wants to keep a close eye on their overall GPA. Let's add a method to the Student class that computes and returns their overall GPA based on the classes that they have taken.

<u>Course</u>	<u>Grade (QP)</u>	<u>Credits</u>	<u>Points</u>
GCIS-123	B+ (3.33)	4	13.32
SWEN-101	A (4.0)	1	4.0
MATH-181	C- (1.67)	3	5.01
PSYC-101	B (3.0)	3	9.0
PHIL-101	C+ (2.33)	3	6.99
Total		14	38.32
GPA			2.74

- Open your `students` module and update the `get_gpa` method.
  - Calculate the student's GPA according to formula that we went over previously.
    - Iterate through the list of courses that the student has taken.
    - Get the points and credits from each course.
    - Use them to compute the overall GPA for the student.
  - Is the `gpa` field needed anymore?
- Print your student's GPA in the `main` function in the `students_main` module.

# Review: Classes & Objects

- A **class** is a **blueprint** or **template** that defines the characteristic of some category of things.
  - **Instances** or **objects** of the class are created by invoking the **constructor**; a special function with the same name as the class.
- A list of **\_\_slots\_\_** may be specified as a **static field** to list the valid names of any instance fields.
  - This **does not** automatically add fields with those names to instances of the class!
  - Each field must still be added using **dot-notation**.
  - Trying to add a field with a name that is not in the **\_\_slots\_\_** list will raise an **AttributeError**.
- An object acts like a container that holds its fields together in one place.
  - In computing, this is referred to as **encapsulation**.
- When implemented properly, encapsulation means that an object **protects** and **controls** access to its fields.
  - Fields are made **private** so that they cannot be directly accessed or changed by other parts of the program.
  - In Python this is accomplished by naming the fields with a **double underscore prefix** (**\_\_**).

A **class** is declared using the class keyword followed by it's name.

The **\_\_slots\_\_** list specifies the **valid** names of any fields that may be attached to an instance. The **double underscore** (**\_\_**) marks fields as **private**.

```
1 class Bicycle:
2     __slots__ = ["__color", "__gears",
3                 "__seat", "__training_wheels"]
4
5     bike = Bicycle()
6     bike.__gears = 1
7     bike.__training_wheels = True
8     print(bike.__training_wheels)
9     bike.__streamers = True
```

An instance of the class is created using the **default constructor** that is provided by Python.

Fields are added to an instance using **dot-notation**, but trying to access private fields from outside of the class will raise an **AttributeError**.

# 11.13 A Car Class

In order to get more practice with methods and special methods, we'll need a new class to play with! Let's write a class to represent a Car with several fields to represent make, model, year, and so on.



Use any one of the classes you've written in this or the previous unit for examples.

- Create a new Python module in a file named `"cars.py"` and define a new class to represent a `Car`.
  - Use `__slots__` to define fields for the car's unique `VIN`, `make`, `model`, `year`, `mileage`, and `fuel`.
  - Don't forget to use proper encapsulation; private fields should all be named starting with a `double underscore` (`__`).



# Review: Constructors, Accessors, & Mutators

A **custom constructor** is named `__init__` and declares at least one parameter: `self`. It is used to initialize fields in a new instance of the class.

```
1 class Bicycle:
2     __slots__ = ["__color", "__gears",
3                 "__seat", "training_wheels"]
4
5     def __init__(self, color, seat):
6         self.__color = color
7         self.__gears = 21
8         self.__seat = seat
9         self.__training_wheels = False
10
11     def get_gears(self):
12         return self.__gears
13
14     def set_color(self, color):
15         self.__color = color
```

**Accessors** are functions that return the value of a private field. **Mutators** are functions that change the value of a private field. Both are only written **as needed**.

- The developer may write a **custom constructor** named `__init__` that declares at least one parameter: `self`; a reference to the object being constructed.
  - Any number of additional parameters may also be declared and used to initialize fields.
- **Private** fields are made accessible using a method named with a `"get_"` prefix that returns the value of the field.
  - A `self` parameter is declared that refers to the object that contains the field that will be returned, e.g. `get_gears(self)`
  - These are called **accessors** or **getters**.
- **Private** fields are made **changeable** using a method named with a `"set_"` prefix.
  - A `self` parameter is declared that refers to the object that contains the field that is being modified, e.g. `set_color(self, color)`
  - A second parameter is declared for the value being used to modify the field.
  - These are called **mutators** or **setters**.

## 11.14 Constructing Cars

Each time an instance of the `Car` class is created, we should guarantee that each of its fields have been initialized with an appropriate value. Add an initializing constructor to the `Car` class now.



- Open your `cars` module and add a **constructor** to the `Car` class.
  - Declare parameters for `self`, `vin`, `make`, `model`, and `year`.
  - The `mileage` and `fuel` should start at 0.
- Add a function named `"print_car"` that prints *all* of the Car's fields to standard output.
- Create a new Python module in a file named `"cars_main"` and add a `main` function that creates and prints at least **two** cars
  - Don't forget to `import cars`.

# Review: Methods

- Up to this point we have exclusively focused on the **state** that classes **encapsulate**.
  - The fields that are contained by objects of the class.
- But classes may also encapsulate **behavior**.
  - Functions can be added to the class.
  - Every function declares at least one parameter that refers to the object on which the function is being called: **self**.
  - The functions may also declare other parameters as needed.
- A function that belongs to an object is called a **method**.
  - A method **should** interact with the fields in the object in some way, e.g. by changing or using them.
  - Otherwise, why is the method in the class?

A **method** is declared inside of a class and the first parameter is always **self** - a reference to the object on which the method is being called.

```
1 class Bicycle:
2     # __slots__ and constructor not shown
3
4     def ride(self, name):
5         print(name, "rides their",
6               self.__color, "bicycle with",
7               self.__gears, "gears and a",
8               self.__seat, "seat!")
9
10        if self.__streamers:
11            print("With streamers!")
```

A **method** should **interact** with the fields in an object in some way. Otherwise, it may not need to be part of the class.

## 11.15 Fill'er Up

If a function does not need to use the fields inside of a class, then it may not need to be a part of the class. The methods in a class should use or manipulate the fields in some way. Let's add a function that adds fuel to a car's gas tank.

```
class Bicycle:

    def ride(self, name):
        print(name, "rides their",
              self.__color, "bicycle with",
              self.__gears, "gears and a",
              self.__seat, "seat!")

    if self.__streamers:
        print("With streamers!")
```

This example method uses several of the fields inside the Bicycle class.

- Open your `cars` module and add a method named `"filler_up"` that declares a parameter for `gallons` of gas.
  - The function should add the specified number of `gallons` to the car's `fuel`.
  - **Challenge:** do not allow the Car to fill with more than `15` `gallons` of fuel.
- Test your new function from `main` in your `cars_main` module.
  - Add some gas to both of your cars and print their fuel level afterwards.

# 11.16 Driving in Cars

Most class have more than one method. Some have many more! Let's add another method that cane be used to "drive" a car. It should change the car's mileage and use up some of its fuel. What should we do if the car runs out of gas?



If you need to, refer to some of the other methods that you wrote in this unit for examples.

- Open your `cars` module and add a method to your `Car` class named "`drive`" that declares a parameter for a distance in `miles`.
  - Add the specified number of `miles` to the car's `mileage`.
  - Subtract an amount of `fuel` based on fuel consumption of **30 miles per gallon**.
  - **Challenge:** do not let the car drive on an empty tank. This includes driving a distance farther than the current amount of fuel would allow, e.g. 150 miles on 4 gallons of fuel.
- Test your new function from the `main` function in the `cars_main` module.
  - Be sure to print each car's mileage and fuel level before and after driving.

## 11.17 Printing Cars?

It would be nice to have an easy way to print all of the important attributes of a car to standard output whenever we need to. Let's remind ourselves what happens when we print an object to standard output...



Google says that this image is a 3D-printed car and you should always believe everything that you see on the internet.

- Open your `cars_main` module and update your `main` function to:
  - Use the `str()` function to translate one of your cars into a string and print it.
  - Pass the other car directly into the `print()` function without using `str()`.
  - What happens?

# Special Methods:

## `__repr__` and `__str__`

The `__str__` function is **automatically** called on an object when it is passed into the `print()` function or used with the string constructor (`str()`).

```
1 class Bike:
2
3     def __str__(self):
4         return "A " + str(self.__color) + " bike."
5
6     def __repr__(self):
7         return "Bike:\n  color=" + self.__color \
8             + "\n  gears=" + str(self.__gears) \
9             + "\n  seat=" + self.__seat \
10            + "\n  training wheels=" + \
11            str(self.__training_wheels)
```

The `__repr__` function is called when an object is passed into the built-in `repr()` function, which returns a string. It is also called if the class doesn't implement the `__str__` function.

- There are many special methods in Python. We have already used one of them.
  - `__init__` is the special name used for custom constructors.
- Python has **many** other special method names. These functions are automatically called under certain circumstances.
- So far, when printing objects, we have made custom print functions, e.g. `print_car`.
  - But wouldn't it be better if we could just pass the object into the `print()` function?
- The `__str__` method is a specially named method that is used to return a nicely formatted string suitable for printing.
  - This is called on an object when the object is passed into the `print()` function.
  - It is also called when using `str()` to construct a string.
- The `__repr__` method is another specially named method that is used to return a very detailed string representation of an object.
  - This is useful for **debugging**.
  - It is called when an object is passed into the built-in `repr()` function.
  - It is also called if the class **doesn't** implement `__str__`.
- Python data structures like lists and sets are exceptions - `repr()` is called for each element when the data structure is passed to the `print()` function.

# 11.18 Represent

The specially named `__repr__` method is called to convert an object that is passed into the built-in `repr()` function into a string. It is also sometimes called if the object does not have a `__str__` method. Let's add a `__repr__` method to the `Car` class!

```
def __repr__(self):  
    return "Bike:" \  
        + "\n  color=" + self.__color \  
        + "\n  gears=" + str(self.__gears) \  
        + "\n  seat=" + self.__seat \  
        + "\n  training wheels=" + \  
        str(self.__training_wheels)
```

The `__repr__` function is used to create a detailed string representation of an object that is suitable for printing or debugging.

- Open your `cars` module and add a `__repr__` method to the `Car` class.
  - Return a **detailed string** representation of the `Car` including the current value of **every field** on a separate line.
  - To be clear, this should **return** a string, **not** print one.
- What happens when you run the `main` function in your `cars_main` module now?



# 11.19 Stringifying

The `__str__` method is called on an object when it is passed into the built-in `str()` function or to convert the object into a string when it is passed into the `print()` function. Let's add a `__str__` function to the `Car` class now!

`__str__` creates strings that are more compact.



```
class Bike:
    def __str__(self):
        return "A " + str(self.__color) + \
            " bike."
```

The `__str__` method may be used to create a more brief/less detailed string representation of an object.

- Open your `cars` module and add a `__str__` method to your `Car` class.
  - Return a **compact, one-line string** representing the car with at least the **VIN**, **make**, and **model**. You may include other details if you'd like.
  - To be clear, the function should **return** a string, **not** print one.
- What happens when you run the `main` function in your `cars_main` module now?

- As mentioned previously, Python includes **many** special methods that are automatically invoked under certain circumstances.
- The `__eq__` method is invoked whenever the `==` operator is used to compare two values to each other.
  - It declares a parameter for `self` and the `other` value.
  - It returns `True` if both parameters are considered equal to each other. It's up to the developer to decide what this means!
  - The other value may not be an instance of the same class!
- Two instances of the same class can see each other's private fields and **compare** them.
- The built-in `type()` function can be used to verify that the other value is the **same type**.
  - e.g `if type(self) == type(other)`
  - This will avoid raising an `AttributeError` when trying to access fields in the other class.
  - If the value is the **wrong type**, the function usually just returns `False`.
- The `__ne__` method is invoked when the `!=` operator is used.
  - It also compares two values but returns `True` if they are **not** equal.
  - It usually just **negates** the value returned from `__eq__`

# Special Methods:

## `__eq__` and `__ne__`

The `__eq__` method is **automatically** called on an object when it is compared to another object using `==`.

The built-in `type()` function can be used to verify that the other value is the same type **before** trying to access its fields. This avoids an `AttributeError`.

```
1 class Bike:
2
3     def __eq__(self, other):
4         if type(self) == type(other):
5             return self.__color == other.__color
6         else:
7             return False
8
9     def __ne__(self, other):
10        return not self.__eq__(other)
```

The `__ne__` method usually negates the value returned by the `__eq__` function.

# 11.20 Car Equality

The `__eq__` method is automatically called on an object whenever it is compared to some other value using the `==` operator. Let's add an `__eq__` method to the `Car` class.



```
def __eq__(self, other):  
    if type(self) == type(other):  
        return self.__color == other.__color  
    else:  
        return False
```

The other value could be **any** type in Python!  
We need to make sure that it **is** a `Car` before  
we try to use dot-notation to access its fields!

- Open your `cars` module and add an `__eq__` method to your `Car` class.
  - It should declare parameters for `self` and the `other` value.
  - Return `True` *iff* the other value **is** a car and both cars have the **same VIN**.
- Create a **third car** in the `main` function of your `cars_main` module.
  - Make sure that it has the **same VIN** as one of the other cars.
  - Print the results of using `==` to compare all three cars to each other.

## 11.21 Sorting Cars

We know that Python automatically sorts lots of types including strings, integers, floats, and even tuples! What happens when we try to sort a list full of cars? Let's find out!



What does it even mean to sort a list of cars?

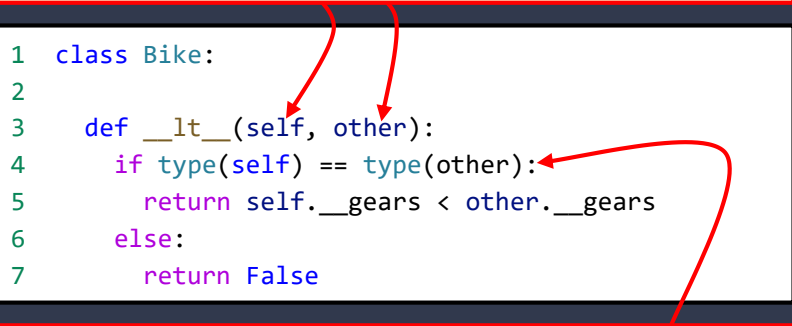
- Open your `cars_main` module and modify the `main` function so that it creates a list of **at least 5 different cars**.
  - Use the three cars that you already have, and add at least two more.
  - **Sort** the list and print it.
- What happens when you **run** the module?

# Special Methods: Relational Operators

Each of the relational methods is called on an object when the corresponding operator is used to compare it to another object using a relational operator.

References to both objects are passed in as the `self` and `other` parameters.

```
1 class Bike:
2
3     def __lt__(self, other):
4         if type(self) == type(other):
5             return self.__gears < other.__gears
6         else:
7             return False
```



The **built-in** `type()` function should be used to prevent an `AttributeError` when accessing fields.

It is up to the developer to decide what each comparison means, e.g. what does it mean for one car to be "less than" another?

- Trying to sort a list of `Car` objects causes a `TypeError`.
  - This is because the sort function tries to **compare** the cars using `<`, and this is not supported by default.
- Yet another set of special methods are automatically called when two objects are compared using the **relational operators**.
  - Each special method declares two parameters:
    - `self` is a reference to the object on which the method is being called.
    - The `other` is a reference to the object to which `self` is being compared.

Comparison Operator	Example	Function	Returns
<	a < b	<code>__lt__</code>	True if a < b
<=	a <= b	<code>__le__</code>	True if a <= b
>	a > b	<code>__gt__</code>	True if a > b
>=	a >= b	<code>__ge__</code>	True if a >= b

# 11.22 Relationships With Cars

Being able to tell whether or not two cars are equal to each other is great! Unfortunately, it doesn't tell us anything about how they should be arranged into sorted order. Let's implement some of the inequality methods so that we can sort cars into a natural order based on their VIN



```
def __lt__(self, other):
    if type(self) == type(other):
        return self.__gears < other.__gears
    else:
        return False
```

Just like the `__eq__` method, we don't want to try to access the fields in the other object until we're sure that it's a `Car`!

- Open your `cars` module and implement each of the following functions in your `Car` class:
  - `__lt__` compares VINs using `<`, e.g. `self.__vin < other.__vin`
  - `__le__` compares VINs using `<=`
  - `__gt__` compares VINs using `>`
  - `__ge__` compares VINs using `>=`
- Run your `cars_main` module again.
  - What happens this time?
- Recall that passing a list or set to `print()` will call `repr()` for each element and not `str()`.
  - Why? Because Guido said changing it to call `str()` would cause [too much disturbance too close to the beta](#).

## 11.23 A Set of Cars

We know if two cars are equal to each other, so we should be able to add them to a set without worrying about adding the same car twice, right? Let's try it and see what happens!



- Open your `cars_main` module and modify the `main` function so that it adds all of the cars in your list to a `set`.
  - Create a set by passing your list to the `set()` function.
  - Print the set of `unique cars`.
- What happens when you `run` the module?


All a set needs to know is if the values are unique, right? RIGHT?

# Special Methods: `__hash__`

- Trying to add an instance of the `Car` class to a set causes a `TypeError` because Python classes are not **hashable** by default.
  - This means that a `Car` won't work as a **key** in a **dictionary**, either.
- In Python, each object is responsible for computing its **own** hash code.
  - This is done by implementing the special `__hash__` method in the class.
  - As usual, a `self` parameter is declared and used to hash a reference to the **object that is being hashed**.
  - The method should **return an integer value**.
  - The value may be **negative**.
- This method is **automatically** called when an object is passed into the **built-in** `hash()` function.

Once the `__hash__` method has been implemented in a class, objects of the class become **hashable**.

```
1 class Bike:
2
3     def __hash__(self):
4         return hash(self.__seat) * self.__gears
```



Remember that the `__hash__` function must be **fast**, **consistent**, and **minimize collisions**.

One way to easily accomplish this is by using the **built-in** `hash()` function to hash some or all of the fields in the object.

**Remember:** **don't** use fields that will **change value** in the hash code. The hash code **must not change**!



## 11.24 Hashing Cars Redux

If a class implements the `__eq__` method it cannot be used inside of a hashing data structure unless it also implements the `__hash__` method. If we want our cars to work in sets and dictionaries, we will need to implement both. Let's do that now!

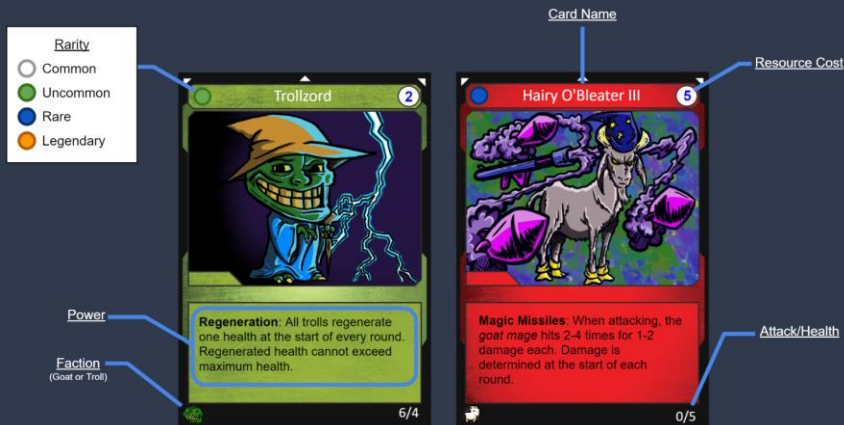


```
def __hash__(self):  
    return hash(self.__seat) * self.__gears
```

The `__hash__` method is called whenever an object is used in a set or dictionary.

- Open the `cars` module and add a `__hash__` function to the `Car` class.
  - The car's VIN should never change, so it should be safe to use it to generate a hash code. Use the built-in `hash()` function to return the hash of the car's VIN.
- What happens when you **run** the `cars_main` module *now*?

# Methods



We will begin implementing a **collectible card game** that pits a heroic team of Goats against legions of vicious Trolls.

We will combine much of what we have learned about classes, objects, encapsulation, constructors, methods, and problem analysis to do this.

- So far in this unit we have learned a lot about **methods**.
  - A method is a function defined by a class.
  - It is called on an object; the object is passed in as an argument to the first parameter (`self`).
  - It should access or change the fields in the object in some way.
- We also learned a lot about **special methods** in python.
  - The `__init__` method is used to initialize a new object.
  - The `__str__` and `__repr__` methods are used to create a string representation of an object.
  - The `__eq__` and `__ne__` methods compare objects for equality and inequality.
  - The `__lt__`, `__le__`, `__gt__`, and `__ge__` methods implement relational operators.
  - The `__hash__` method is used to generate a hash code usable in sets and dictionaries.
- Next, we will practice a little bit of everything that we have learned about classes, objects, & methods over the last two units to build a software system.

# Goats vs. Trolls the Collectible Card Game™

A collectible card game (CCG) allows players to simulate an epic battle by playing cards against each other.

In most CCGs, the cards played represent fantasy heroes of some kind, each of which has health and attack ratings that are used to damage other cards and/or players.

Goats vs. Trolls is one such game that is very similar to [Blizzard's Hearthstone](#), though it is also *much* simpler.

In a game of Goats vs. Trolls, each player chooses to represent one of two possible factions: the heroic Goats, or the villainous Trolls.

The players use the cards in their decks to do battle with each other until one player's score is reduced to 0.



We will start by performing a **problem analysis** using a brief description of the *Goats vs. Trolls* collectible card game.

# Review: Problem Analysis

- It is often the case that your customer will provide you with the requirements for the software that they would like you to build for them written in **long-form language**.
  - This document will be written using the customer's vernacular - the **domain language** that users in their area of expertise use every day.
- Before software development can begin, the problem statement will need to be broken down into a much more simplified form.
- One technique is to perform a **noun/verb analysis**.
  - Make one pass through the document and create a list of all of the nouns.
  - Make a second pass to create a list of all of the verbs.
  - It is important that you keep these terms in the domain language - do not "inject" your own phrases or wording!
- Many of the nouns will end up being **classes** or **attributes** in your software.
- Many of the verbs will end up being **methods** in your classes.
- This process is called **problem analysis**.
  - Sometimes it is also called **domain analysis** or **requirements analysis**.

book

title (string)

author (string)

copies (int)

patron

id (string)

name (string)

checked out books (list)

want list (set)

The goal of a problem analysis is to identify the classes and attributes needed to build a software system that meets your customer's requirements.

# 11.25 Goats vs. Trolls Problem Analysis

Performing a problem analysis is an excellent way to start planning a large, complex software system. Let's analyze the *Goats vs. Trolls* problem statement so that we can begin to design the software system that will implement a playable version of the game.

In a game of Goats vs. Trolls, each player chooses to represent one of two possible factions: the heroic Goats, or the villainous Trolls. The players draw cards from a shuffled deck of 40 cards and place them into a hand. The players take turns and use the cards in their hands to attack the other player's cards until one player's score is reduced to 0.

Each card in a game of GvT has several attributes including:

- Name
- Resource cost
- Faction (Goats or Trolls)
- Rarity (legendary, rare, uncommon, or common)
- Attack power
- Health

- Create a new text file named "gvt\_analysis.txt" and use it to perform a problem analysis using the brief description to the left.
  - Start by creating a comma-separated list of all of the nouns in the problem statement on a single line.
  - Identify which nouns you think will be classes and write the name of each on a separate line.
  - Identify the nouns that you think will be attributes (fields) of a class, and indent them under the corresponding class.
  - If you can, identify the type of each attribute, e.g. string, int, etc. If you think that the attribute will be a data structure, try to choose the one that provides the best performance.
  - Next, list all of the verbs in the problem statement. Try to identify which may end up being methods and indent them under the correct class.

# Review: Classes & Objects

- A **class** is a **blueprint** or **template** that defines the characteristic of some category of things.
  - **Instances** or **objects** of the class are created by invoking the **constructor**; a special function with the same name as the class.
- A list of `__slots__` may be specified as a **static field** to list the valid names of any instance fields.
  - This **does not** automatically add fields with those names to instances of the class!
  - Each field must still be added using **dot-notation**.
  - Trying to add a field with a name that is not in the `__slots__` list will raise an `AttributeError`.
- An object acts like a container that holds its fields together in one place.
  - In computing, this is referred to as **encapsulation**.
- When implemented properly, encapsulation means that an object **protects** and **controls** access to its fields.
  - Fields are made **private** so that they cannot be directly accessed or changed by other parts of the program.
  - In Python this is accomplished by naming the fields with a **double underscore prefix** (`__`).

A **class** is declared using the class keyword followed by it's name.

The `__slots__` list specifies the **valid** names of any fields that may be attached to an instance. The **double underscore** (`__`) marks fields as **private**.

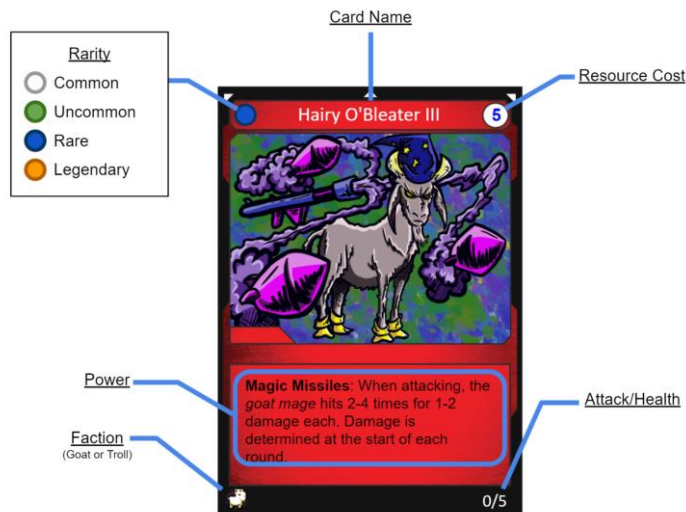
```
1 class Bicycle:
2     __slots__ = ["__color", "__gears",
3                 "__seat", "__training_wheels"]
4
5     bike = Bicycle()
6     bike.__gears = 1
7     bike.__training_wheels = True
8     print(bike.__training_wheels)
9     bike.__streamers = True
```

An instance of the class is created using the **default constructor** that is provided by Python.

Fields are added to an instance using **dot-notation**, but trying to access private fields from outside of the class will raise an `AttributeError`.

# 11.26 A Card Class

Let's refresh our memories regarding how classes are built by starting with a GvT `Card` class and defining slots for each of its attributes.



If you need a reference, take a look at one of the classes you wrote in this or the previous unit.

- A module named "gvt" has been provided to you. Open it and define a new class to represent a `Card`.
  - Use `__slots__` to define fields for the attributes of a GvT card including:
    - Name
    - Resource cost
    - Rarity (Common, Uncommon, Rare, Legendary)
    - Faction
    - Attack Power
    - Health
  - Don't forget to use proper encapsulation; private fields should all be named starting with a **double underscore** (`__`).

# Review: Constructors

A **custom constructor** is named `__init__` and declares at least one parameter: `self`. It is used to initialize fields in a new instance of the class.

```
1 class Bicycle:
2     __slots__ = ["__color", "__gears",
3                 "__seat", "training_wheels"]
4
5     def __init__(self, color, seat):
6         self.__color = color
7         self.__gears = 21
8         self.__seat = seat
9         self.__training_wheels = False
```

The fields in the new object are set using **dot notation** and may use values passed in as parameters, default values, or values that are derived in some way from the other fields or parameters.

- Python provides a **default constructor** that will create an empty object without any fields.
- You may write a **custom constructor** named `__init__` that declares at least one parameter: `self`; a reference to the object being constructed.
  - Any number of additional parameters may also be declared and used to initialize fields.
- The constructor is used to ensure that each and every field in the object that is being constructed has been initialized with an appropriate value.
  - Some may be assigned the value of one of the parameters, e.g. `self.__color = color`
  - Others may use default values, e.g. `self.__training_wheels = False`
  - Others may use a value that is **derived** from some of the other fields or parameters.
- The constructor is invoked from outside of the class using the name of the class, e.g. `bike = Bicycle("red", "banana")`



# 11.27 Constructing Cards

We won't be able to play the game unless we can construct cards! Let's add a constructor to the `Card` class that we can use to make new cards.



- Open your `gvt` module and add a **constructor** to the `Card` class.
  - Declare parameters for each of the card's attributes.
- Create a new Python module named "`gvt_main`" and define a `main` function that creates at least two cards.
  - Use whatever values for the various attributes that you would like.
  - What happens if you try to use dot notation to print the values inside the cards?

# Review: Accessors & Mutators

- We have talked about two different aspects of **encapsulation**.
  - An object acts like a container that holds its state (fields) together in one place.
  - The object protects access to its fields by making them private; in Python this means naming the fields with a double underscore (`__`).
- Private fields cannot be directly accessed by other parts of the program outside of the class.
  - This is a good thing! Imagine the chaos if a Troll could just set the health of a goat to -100, or change its name to "Kick-me Johnson."
- If the developer *wants* to make the value of a field visible outside of the class, they write an **accessor**; a method named with a `get_` prefix that returns the value of the field.
  - This must be used carefully with reference types!
- If the developer *wants* to allow a field to change, they write a **mutator**; a method named with a `set_` prefix that modifies the field.

```
1 class Bicycle:
2     __slots__ = ["__color",
3                 "__gears", "__seat",
4                 "__training_wheels"]
5
6     def get_gears(self):
7         return self.__gears
8
9     def set_color(self, color):
10        self.__color = color
```

**Accessors** ("getters") are named with a "get\_" prefix and return the value of some field.

**Mutators** ("setters") are named with a "set\_" prefix and modify the value of a field.



Accessors are necessary so that other parts of the program can see the value of private fields inside of an object. Mutators are only needed if other parts of the program should be able to *change* the value of a field inside of an object. Let's add accessors and (maybe?) mutators to the card class now.

- Open the `gvt` module and define an accessor for each field.
  - Each accessor should be named with the `get_` prefix followed by the field's name (without the double underscores), e.g. `"get_gears"`.
  - It should declare only one parameter for `self`.
  - It should return the value of the field.
- Define mutators **only** if the field should be modified directly from other parts of the program.
  - Ask yourself: should a Troll be able to call this function on a Goat?
- Update the `main` function in `gvt_main` to call a few of your accessors (and any mutators you may have written) and print the values.

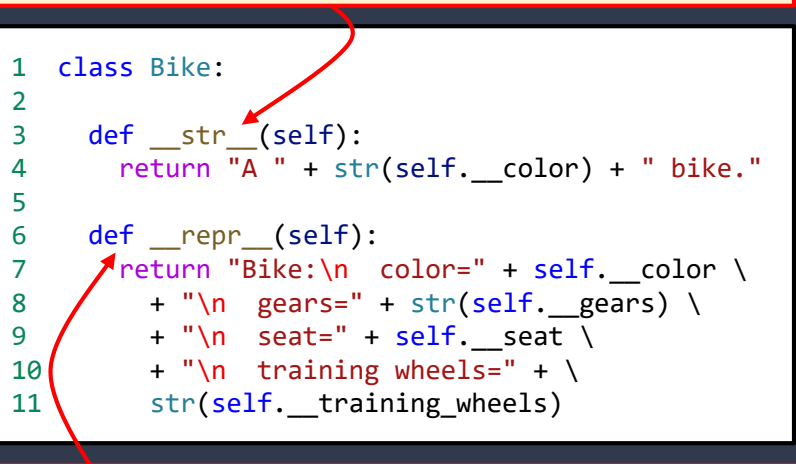
If you need a reference, take a look at one of the classes you wrote in this or the previous unit.

# Review: Special Methods

## `__repr__` and `__str__`

The `__str__` function is **automatically** called on an object when it is passed into the `print()` function or used with the string constructor (`str()`).

```
1 class Bike:
2
3     def __str__(self):
4         return "A " + str(self.__color) + " bike."
5
6     def __repr__(self):
7         return "Bike:\n  color=" + self.__color \
8             + "\n  gears=" + str(self.__gears) \
9             + "\n  seat=" + self.__seat \
10            + "\n  training wheels=" + \
11            str(self.__training_wheels)
```



The `__repr__` function is called when an object is passed into the built-in `repr()` function, which returns a string. It is also called if the class doesn't implement the `__str__` function.

- There are many special methods in Python. We have already used one of them.
  - `__init__` is the special name used for custom constructors.
- Python has **many** other special method names. These functions are automatically called under certain circumstances.
- So far, when printing objects, we have made custom print functions, e.g. `print_car`.
  - But wouldn't it be better if we could just pass the object into the `print()` function?
- The `__str__` function is a specially named function that is used to return a nicely formatted string suitable for printing.
  - This is called on an object when the object is passed into the `print()` function.
  - It is also called when using `str()` to construct a string.
- The `__repr__` function is a specially named function that is used to **return** a very detailed string representation of an object.
  - This is useful for **debugging**.
  - It is called when an object is passed into the built-in `repr()` function.
  - It is also called if the class **doesn't** implement `__str__`.
- Python data structures like lists and sets are exceptions - `repr()` is called for each element when the data structure is passed to the `print()` function.

## 11.29 Represent Redux

In Python the built-in `repr()` function is used to make a detailed string representation of an object that is useful for printing, but it only works if the object's class implements the `__repr__` method. Add a `__repr__` function to your `Card` class that includes all of a card's various details

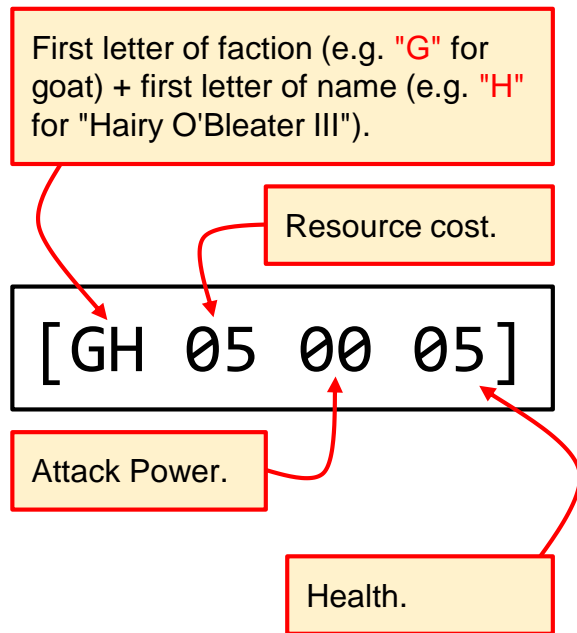
```
Hairy O'Bleater III
Rarity: R
Faction: Goat
Resource Cost: 5
Attack Power: 0
Health Points: 5
```

The `__repr__` function is called when the object is passed into the built-in `repr()` function, printed inside of a list or a set, or if the `__str__` method is not implemented.

- Open your `gvt` module and add a `__repr__` method to the `Card` class.
  - Return a **detailed string** representation of the `Card` including the current value of **every field** on a separate line. Use the example to the left as a reference.
  - To be clear, this should **return** a string, **not** print one.
- Modify the `main` function in your `gvt_main` module to use the built-in `repr()` function to convert one of your cards to a string and print it.

# 11.30 Stringifying

The `str()` function can be used to convert an object into a compact string suitable for printing; it is automatically called when an object is passed into the built-in `print()` function. But it only works if the object's class implements the `__str__` method. Add a `__str__` method to your `Card` class that returns a compact version of the card's details.



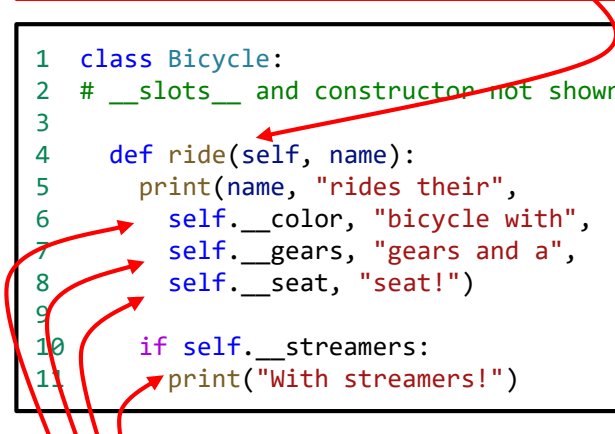
- Open your `gvt` module and add a `__str__` method to your `Card` class.
  - Return a **compact, one-line string** representing the card that matches the format suggested on the left.
  - You can use something like `"{:02d}".format(self.__health)` to add leading zeros to your strings.
  - To be clear, the function should **return** a string, **not** print one.
- Update the `main` function in your `gvt_main` module to print each card on the same line.

# Review: Other Methods

- Up to this point in this exercise we have focused mostly on the **state** that classes **encapsulate**.
  - The fields that are contained by objects of the class.
  - Constructors to initialize those fields.
  - Accessors to view and mutators to modify those fields.
- But classes may also encapsulate **behavior**.
  - Functions in the class that perform some kind of computation that interacts with the fields in the class.
  - Every function declares at least one parameter that refers to the object on which the function is being called: **self**.
  - The functions may also declare other parameters as needed.
- A function that belongs to an object is called a **method**.
  - These functions use or manipulate the fields in the class.
  - They are invoked on an object using dot notation.

A **method** is declared inside of a class and the first parameter is always **self** - a reference to the object on which the method is being called.

```
1 class Bicycle:
2     # __slots__ and constructor not shown
3
4     def ride(self, name):
5         print(name, "rides their",
6               self.__color, "bicycle with",
7               self.__gears, "gears and a",
8               self.__seat, "seat!")
9
10        if self.__streamers:
11            print("With streamers!")
```



A **method** should **interact** with the fields in an object in some way. Otherwise, it may not need to be part of the class.

# 11.31 Attaaaaaack!

Methods are functions that are added to a class and use the fields inside the class to complete some task. Let's practice by adding some methods to the card class that can be used to facilitate combat between two or more cards.



- Open your `gvt` module and define a method named "`damage`" that declares a parameter for the `amount` of damage.
  - Subtract the amount of damage from the card's health.
  - Do not let the health drop below 0!
  - Return any "excess" damage, e.g. if the damage is 10 and the card's health is 4, return  $10 - 4 = 6$ .
- Define another method named "`is_conscious`".
  - Return `True` if the card has 1 or more health remaining.
  - Return `False` if the card's health is 0.
- Test your new function from `main` in your `gvt_main` module.
  - Attack each of your cards and print them afterwards.



- Another special method is the `__eq__` method, which is invoked whenever the `==` operator is used to compare two values to each other.
  - It declares a parameter for `self` and the `other` value.
  - It returns `True` if both parameters are considered equal to each other. It's up to the developer to decide what this means!
- Two instances of the same class can see each other's private fields and **compare** them.
- The built-in `type()` function can be used to verify that the other value is the **same type**.
  - e.g `if type(self) == type(other)`
  - If the value is the **wrong type**, the function usually just returns `False`.
- Yet another set of special methods are automatically called when two objects are compared using the **comparison operators**.
  - The `__lt__` method is one such method that declares two parameters:
    - `self` is a reference to the object on which the method is being called.
    - The `other` is a reference to the object to which `self` is being compared.
    - It returns `True` if `self` is less than `other` and can be used for sorting.

# Review: Special Methods

## `__eq__` and `__lt__`

The `__eq__` method is **automatically** called on an object when it is compared to another object using `==`. The **built-in** `type()` function should be used **before** trying to access the private fields in the other object.

```
1  def __eq__(self, other):
2      if type(self) == type(other):
3          return self.__color == other.__color
4      else:
5          return False
6
7  def __lt__(self, other):
8      if type(self) == type(other):
9          return self.__gears < other.__gears
10     else:
11         return False
```

The `__lt__` method is called when objects in a list are sorted. It returns `True` if `self` comes before `other` in sorted order.

# 11.32 Card Equality

Two cards are considered equal *iff* they have the same attributes *other than the health and the name*. Add an `__eq__` method to your Card class.



==



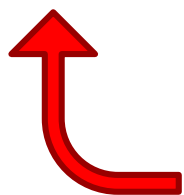
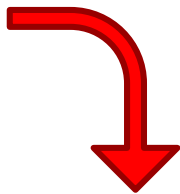
?

```
def __eq__(self, other):  
    if type(self) == type(other):  
        return self.__color == other.__color  
    else:  
        return False
```

- Open your `gvt` module and define an `__eq__` method in your `Card` class.
  - It should declare parameters for `self` and the `other` card.
  - Return `True` *iff* the cards have the **same attributes other than the health and the name**.
- Create a **third card** in the `main` function of your `gvt_main` module.
  - Make sure that it has the **same attributes** as one of the other cards with a different name.
  - Print the results of using `==` to compare all three cards to each other.

## 11.33

## Sorting Cards



Remember: printing a list of cards will call `__repr__` on each card.

CCG players usually like to sort the cards in their hands to help them make decisions about which cards to play next. There are many possible ways to sort GvT cards, but let's assume that we want to sort them by resource cost from lowest to highest. If two cards have the same resource cost, sort them by name.

- Open your `gvt` module and define an `__lt__` function in the `Card` class.
  - Declare parameters for `self` and the `other` card.
  - Return `True` if the resource cost is less than the other card.
  - If the resource costs are the same, compare the names of the cards.  
*Hint: use `<` to compare the names.*
- In the `gvt_main` module, modify the `main` function to add all of your cards to a list, sort it, and print it out.
  - If necessary, add new cards with the same resource cost but different names.

# 11.34 Making Cards

Cards in Goats vs. Trolls (GvT) have a different allocation of health points, attack power, and resource cost depending on rarity.

Rarity	Health/Attack Points (randomly distributed)	Resource Cost (random)
Common	8	1 - 3
Uncommon	12	2 - 5
Rare	16	4 - 7
Legendary	24	10

The minimum health points for any card is 1. Otherwise the values should be distributed randomly between health and attack points. For example, an Uncommon card may have 3 health points and 9 attack points ( $3 + 9 = 12$ ).

- Open your `gvt` module and define a new function named `make_card` that declares parameters for `faction` and the `rarity`.
  - If the card represents the Goat faction, the names should be generated using the provided `goatils` module.
  - Trolls should have more generic names such as "Troll," "Trollzord," "Trolling," etc. You may choose how to assign troll names, e.g. choosing them randomly from a list.
- Use the table on the left to set the health, resource, and attack power.
- Return the card.
- In `main` in the `gvt_main` module, create five different cards and print them.

## 11.35 Making Decks

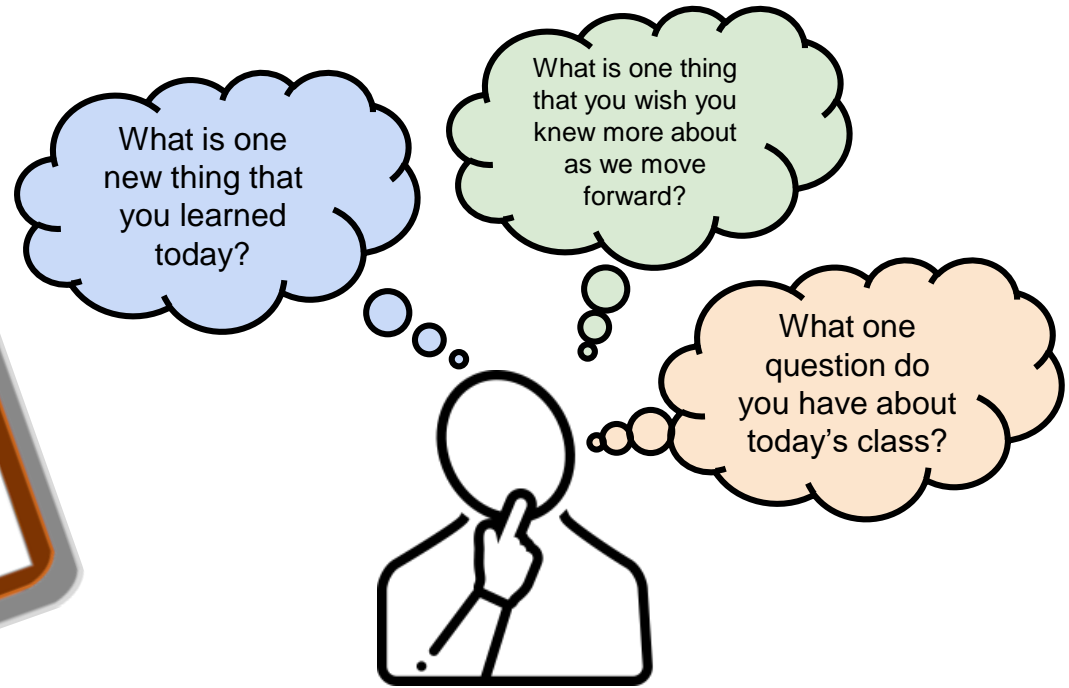
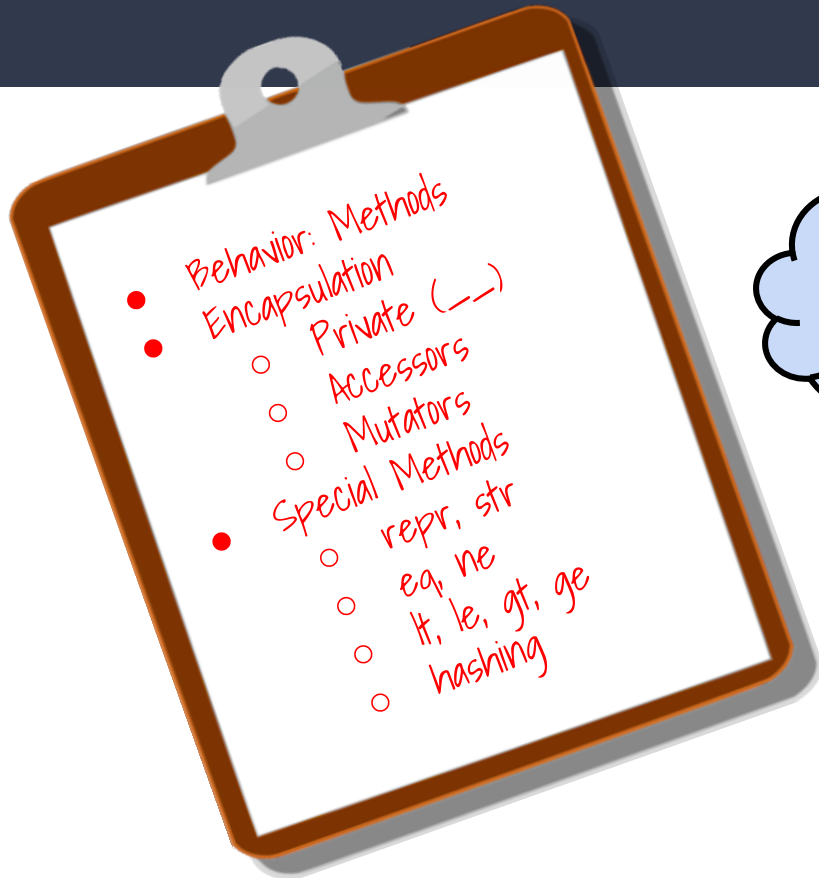
Each player in a game of Goats vs. Trolls starts with a deck of 40 cards: 20 common, 10 uncommon, 8 rare, and 2 legendary.



- 20 common
- 10 uncommon
- 8 rare
- 2 legendary

- Define a function in the `gvt` module that makes a deck of cards and returns it.
  - Declare a parameter for `faction`.
  - Make sure that the deck is shuffled before you return it!
- In your `main` function in the `gvt_main` module, make a deck of trolls and another for goats.

# Summary & Reflection



Please answer the questions above in your notes for today.