# CS134 Lecture: Introduction to Classes & Objects

# Announcements & Logistics

- **Lab 7 due** tonight/tomorrow at 10 pm
  - Make sure your **images and values** match the handout
- **HW 7** will be released today, due Monday at 10 pm
- Lab 8 is also a partner lab:
  - Fill out partner google form (from Lida) by **tomorrow @ noon**
  - Every student has to fill out the form **(both partners)**
  - **Must attend one lab session together**
  - Mon lab due on Wed, Tue lab due on Thur

**Do You Have Any Questions?**

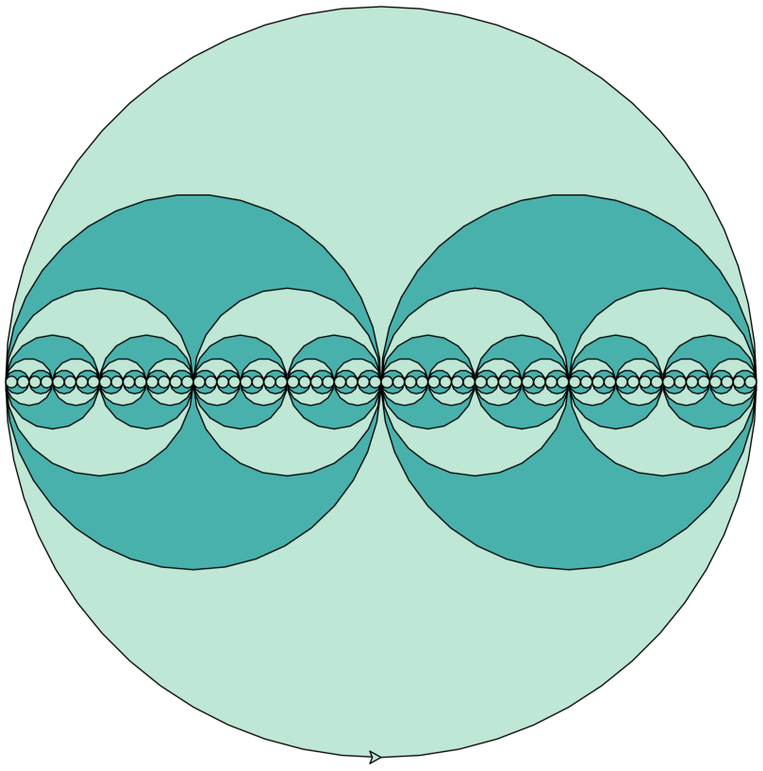
# Pair Programming: Best Practices

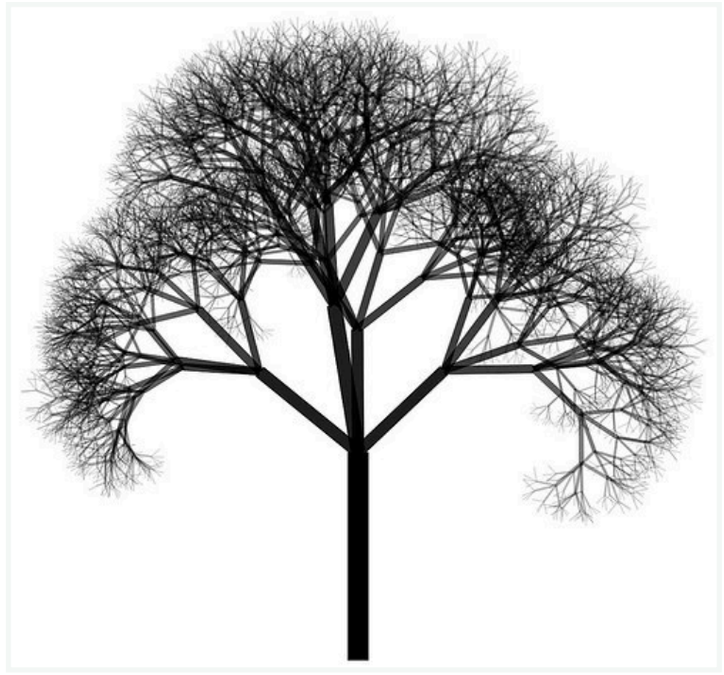
- Goal is to work together as a team using **one computer**:
  - Driver: controls the keyboard (handles the details, debugging, etc)
  - Navigator:  helps guide (big picture person, decides where to go)
- **Both roles are important** and you both should switch roles often!
- **Communication is key**:  but be polite, respectful and patient

- Discussing high level strategies and goals early, before writing the code, helps avoid bugs as well as conflict
- Resources:
  - How Pair Programming Really Works by Stuart Wray
  - Longish article on pair programming pros, cons, and strategies

# Last Time

- Graphical examples of recursion,  important takeaways:
  - how to break down a problem recursively
  - maintaining invariance between function calls
- Review leftover tree example (similar to shrub question in lab)





# Recursion:  Wrap Up

- Why choose recursion over iteration?
  - Some problems have a ***natural recursive structure***
  - Using recursion on them leads to elegant and concise solutions
  - Fewer lines of code often correlates with less debugging!
- We will use recursion to search and sort in a few weeks
- Recursion also helps us build and maintain complex data structures
- Downsides: Recursive approaches *can* have efficiency overhead
    - Steeper learning curve (but can be very rewarding once you get the hang of it)
    - To understand recursion you must understand recursion…

# Today

- Start discussing our next topic: **classes** and **objects**
  - Python is an **object oriented programming** (OOP) language
  - Everything in Python is an **object** and has a **type**
- Learn how to define our own **classes** (**types**) and **methods**

# Objects in Python

# Objects in Python

- We have seen many ways to store data in Python

```
1234 3.14159  "Hello" [1, 5, 7, 11, 13] {1, 2, 3}
{"CA": "California", "MA": "Massachusetts"}
```

- Each of these is an **object**, and every object in Python has:
    - a **type** (int, float, string, list, tuples, dictionaries, sets, etc)
        - an internal **data representation** (primitive or composite)
        - a set of functions/methods for **interacting** with the object
- Vocabulary: A *specific object* is an **instance** of a type
    - `1234` is an instance of an **int**
    - `"Hello"` is an instance of a **string**

# type(object)

- The `type()` function returns the data type for an object

```
>>> type(1234)
<class 'int'>
>>> type("hello")
<class 'str'>
>>> type([1, 5, 7, 11, 13])
<class 'list'>
>>> type(range(5))
<class 'range'>
```

# Objects and Types in Python

**Everything in Python is an object and has a type!**

- Even functions are a type!
- Guido designed the language according to the principle "first-class everything"

```
>>> def greeting():
...   print("Hello")
...
>>> type(greeting)
<class 'function'>
```

"*One of my goals for Python was to make it so that all objects were "first class." By this, I meant that I wanted all objects that could be named in the language (e.g., integers, strings, functions, classes, modules, methods, and so on) to have equal status. That is, they can be assigned to variables, placed in lists, stored in dictionaries, passed as arguments, and so forth.*" — ***Guido Van Rossum***
(Blog, The History of Python, February 27, 2009)

# Stepping Back: Object-Oriented Programming (OOP)

- Python is an **"object-oriented" language**
    - We have been hinting at this aspect all semester
    - Today we will embrace it!
- **OOP** (object oriented programming) is a fundamental programming paradigm
- It has four major principles:
    - **Abstraction** - handle complexity by ignoring/hiding messy details
    - **Inheritance** - derive a class from another class that shares a set of attributes and methods
    - **Encapsulation** - bundling data and methods that work together in a class
    - **Polymorphism** - using a single method or operator for different uses
- We'll explore some of these principles in more detail in the coming lectures

# What are Objects?

- It's time to *formally* define **objects** in Python
- Objects are:
  - collections of data (variables or **attributes**) and
  - **methods** (functions) that act on those data
- Example of **abstraction**:
  - Abstraction is the art of hiding messy details
  - Methods define behavior but hide implementation and internal representation of data
    - e.g., append is a method that is applied to a list type: we don't need to know how it works to use it

# Example: `[1,2,3,4]` has type `list`

- We don't really know how Python stores lists internally
- Fortunately we typically don't sweat the inner working of lists every time we use them (we've been doing it all semester!)
- How do we manipulate lists? Using the **operators** or **list methods** provided by Python.
  - `list3 = list1 + list2` (concatenation using an operator)
  - `list1.append("dog")` (adding to a list using the append operator, this mutates `list1` rather than creating a new list)
- Recall that we can also use the + operator for strings! This is an example of polymorphism
- **Take away:** Internal representation of objects should be hidden from users. Objects are manipulated through associated **methods**.

# Example: `[1,2,3,4]` has type `list`

- We don't really know how Python stores lists internally
- Fortunately we typically don't sweat the inner working of lists every time we use them (we've been doing it all semester!)
- How do we manipulate lists? Using the **operators** or **list methods** provided by Python.
  - `list3 = list1 + list2` (concatenation using an operator)
  - `list1.append("dog")` (adding to a list using the append operator, this mutates `list1` rather than creating a new list)
- Recall that we can also use the `+=` operator for strings! This is an example of operator overloading.

This is the same as `list1+=["dog"]` !!!

- **Take away:** Internal representation of objects should be hidden from users. Objects are manipulated through associated **methods**.

# What are Methods?

- Methods are *functions* that operate only on the specific object instance that comes before the **dot notation** with which they are called:

```
>>> l = ['mike', 'and']
>>> l.append('ike')
>>> l
['mike', 'and', 'ike']
>>> l.extend(['snap', 'crackle', 'pop'])
>>> l
['mike', 'and', 'ike',
 'snap', 'crackle', 'pop']
>>> l.reverse()
>>> l
['pop', 'crackle', 'snap',
 'ike', 'and', ‘mike']
```

appends a single item to the end of `list l`

Adds a sequence to the end of `list l`

Reverses the order of the list of `list l`

Some methods have arguments, some don't. Just like functions!

# What are Methods?

Discover more list & str methods with `pydoc3 list` or `pydoc3 str` !

- Methods are *functions* that operate only on the specific object instance that comes before the **dot notation** with which they are called:

```
>>> s = "    CSCI 134 is great!\n \t"
>>> s.lower()
'    csci 134 is great!\n \t'
>>> s.isalpha()
False
>>> s.strip()
'CSCI 134 is great!'
>>> s.strip()[0].isalpha()
True
>>> "a,b,c,d".split(',')
['a', 'b', 'c', 'd']
```

lowercases all characters in the `str`ing `s`

Is the `str`ing `s` made only of letters?

Remove whitespace from left/right sides of the `str`ing `s`

Method calls can be combined, just like function calls!

# Many built-in String & List Methods

```
word = 'Williams College'
word.split()                                  ['Williams','College']
word.upper()                                    'WILLIAMS COLLEGE'
word.lower()                                    'williams college'
word.replace('iams', 'herst')                  'Willherst College'
word.replace('mith', 'herst')                 'Williams College'
```

Returned value

Notice how methods use dot notation

```
word = '   Spacey College   '
word.strip()                                  'Spacey College'
alist = ['Williams', 'College']
' '.join(alist)                                'Williams College'
```

**Important note: Strings are immutable.** None of these operations change/affect the original string. **They all return a new string!**

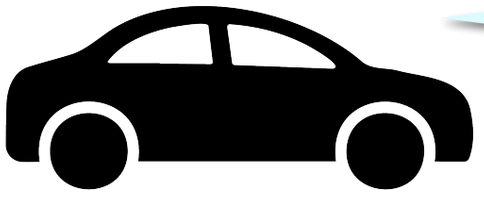
# Defining Our Own Types

# Creating Our Own Types: Classes

- It's time to move beyond just the built in Python objects!
- We can create our own data types by defining our own **classes**
  - Classes are like blueprints for objects in Python
- **Creating** a class involves:
  - Defining the **class name, attributes, methods**
- **Using** the class involves:
  - Creating **new instances** of the class (which create specific objects)
    - ```
      my_list = [1, 2],
      my_other_list = list("abc")
      ```
  - Performing operations on the instances through methods
    - ```
      my_list.append(3)
      ```

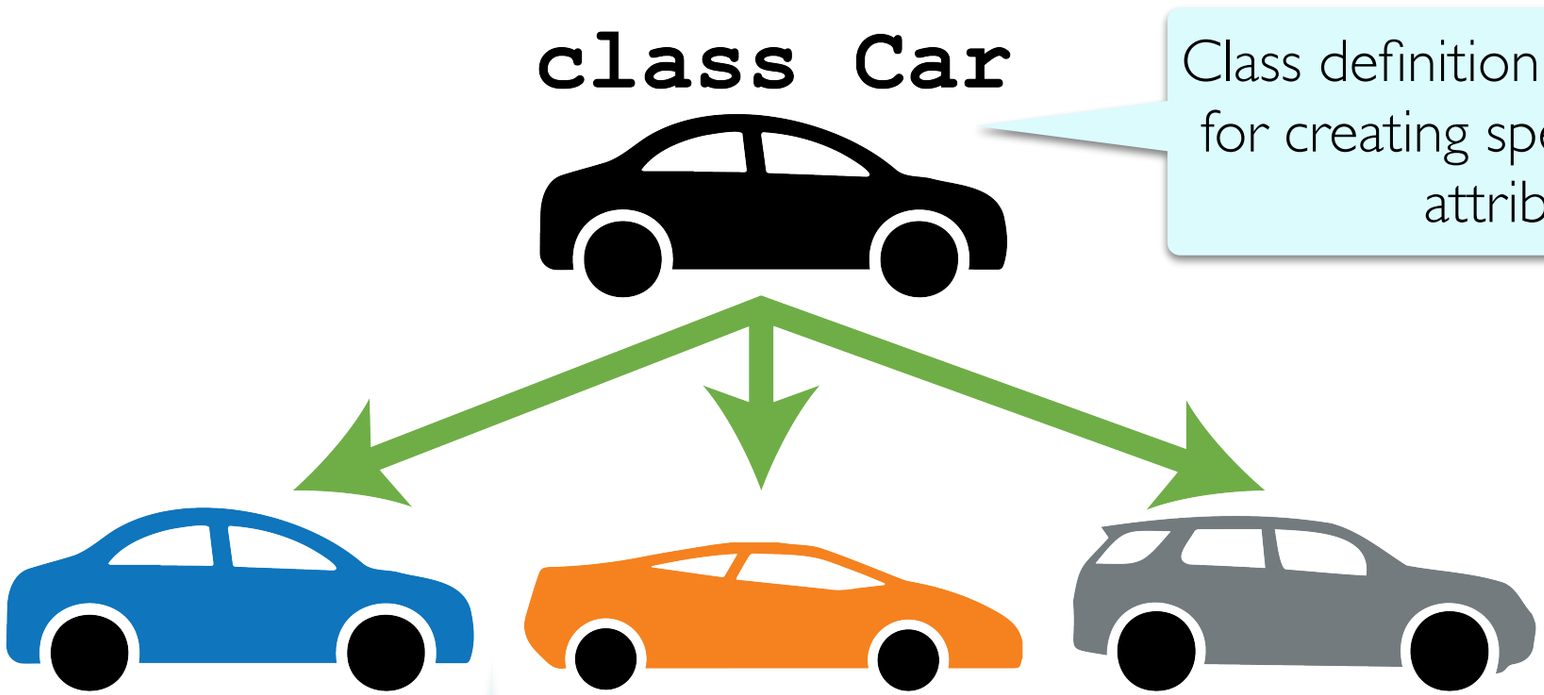
# Defining Our Own Type:  Car class

**`class Car`**



Class definition provides a "blueprint" for creating specific cars and specify **attributes** of cars

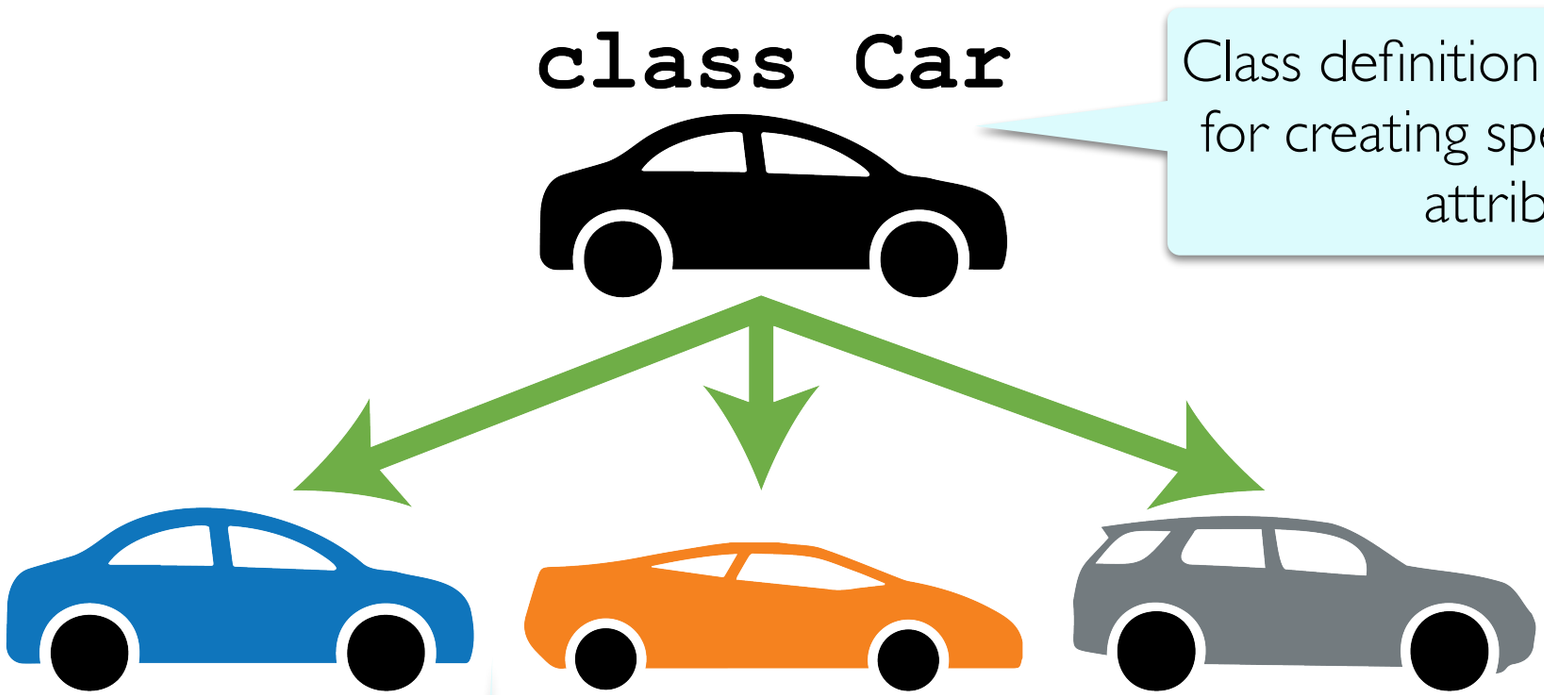
# Defining Our Own Type: Car class




Class definition provides a "blueprint" for creating specific cars and specify attributes of cars

Specific instances of the Car class

# Defining Our Own Type: Car class




Class definition provides a "blueprint" for creating specific cars and specify attributes of cars

_color Blue
_make Toyota
_model Prius

_color Orange
_make Ford
_model Mustang

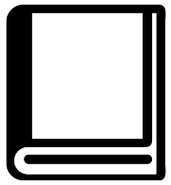
_color Silver
_make VW
_model Golf

Specific instances of the Car class

Providing values for attributes of the Car class, such as color, make, and model, define key features of individual instances

# Defining Our Own Type: Book class

**`class Book`**



Class definition provides a "blueprint" for creating specific books and specify attributes of books

| | | |
|---|---|---|
| `_title` **Fellowship of the Ring** | `_title` **Pride and Prejudice** | `_title` **Parable of the Sower** |
| `_author` **J.R.R. Tolkein** | `_author` **Jane Austen** | `_author` **Octavia Butler** |
| `_year` **1954** | `_year` **1813** | `_year` **1993** |

Specific instances of the Book class

Providing values for attributes of the Book class, such as title, author, and year, define key features of individual instances

# Defining Methods of a Class

- Methods are defined as part of the class definition and describe how to interact with the class objects
- Example: Recall the following methods for the list class

```
>>> lst = list()
>>> lst.extend([1,2,3])
>>> lst
[1, 2, 3]
>>> lst.append(4)
>>> lst
[1, 2, 3, 4]
```

dot notation to "call" the method on the object

# Defining Methods of a Class

- On the previous slide, we called methods like `append()` and `extend()` on a particular list **object** `lst`.
- We can define methods in our classes in a similar way
- Consider this simple example:

Class name (note the use of CamelCase by convention)

```
class SampleClass:
    """Class to test the use of methods"""
    def greeting(self):
        print("Hello")
```

# Defining Methods of a Class

- To create methods that can be called on an instance of a class, they must have a parameter which takes the instance of the class as an argument
- In Python, the **first parameter of a method is always `self`, and is used as a reference to the calling instance**

```
class SampleClass:
    """Class to test the use of methods"""
    def greeting(self):
        print("Hello")
```

**All** methods include **`self`** as the first parameter.

# Our First Method

```
class SampleClass:
    """Class to test the use of methods"""
    def greeting(self):
        print("Hello")
```

- How do we call the greeting method?
- We create an **instance** of the class and call the method on that instance using dot notation:

```
>>> sample = SampleClass()

>>> sample.greeting()
Hello
```

**sample** is an instance of SampleClass

Invoke the `greeting()` method on sample

# Mysterious `self` Parameter

- Even though method definitions have `self` as the first parameter, **we don't pass this parameter explicitly** when we invoke the methods
- This is because whenever we call a method on an object, the object itself is **implicitly** passed as the first parameter
- Note: In other languages (like Java) this parameter is implicit in method definitions but in Python it is explicit and by convention named **`self`**
- **Take away:**
  - When **defining** methods, always include **self**
  - When calling or invoking methods, the value for self is passed implicitly (meaning, we don't specify it, but it happens automatically)

# Mysterious `self` Parameter

- In interactive python:
- *(Recall the id(..) function which returns the identity/address of the object)*

```
>>> class Sample():
...     def __init__(self):
...         print(id(self))

>>> s = Sample()
4404099008

>>> id(s)
4404099008
```

`self` is the instance we are applying the method to!

# Summary of Classes and Methods

- **Classes** allow us to define our own data types
- We create **instances** of classes and interact with those instances using methods
- All **methods** belong to a class, and are defined within a class
- A method's purpose is to provide a way to access/manipulate **instances** of the class)
- The first parameter in the method definition is **the reference to the calling instance (self)**
- When **invoking** methods, this reference is provided implicitly

# Defining Our Own Class: Book

- Key features of a class:
    - **Attributes** that describe instance-specific data
    - **Methods** that act on those attributes
- When defining a new class (aka an object blueprint), it's important to identify what **attributes** are required and what actions will be performed using those attributes (**methods**)
- For example, suppose we want to define a new `Book` class
    - Attributes?
    - Methods?

# Defining Our Own Class: Book

- Key features of a class:
    - **Attributes** that describe instance-specific data
    - **Methods** that act on those attributes
- When defining a new class (aka an object blueprint), it's important to identify what **attributes** are required and what actions will be performed using those attributes (**methods**)
- For example, suppose we want to define a new `Book` class
    - Attributes?
        - Title, author, publication year, genre, …
    - Methods?
        - `same_author_as( ), years_since_pub( ), …`

# Defining Our Own Class: Book

Name of class (always capitalized by convention)

```
class Book:
    """This class represents a book"""
    # indented body of class
    definition
```

**Creating instances of the class:**

```
book1 = Book()
book2 = Book()
```

book1 is an instance of class Book

book2 is another (different) instance of class Book

# Attributes

- Objects have *state* which is typically held in **instance variables** or (in Pythonic terms) **attributes**.
- Example: For our **Book** class, these include the book's title, author, and publication year
- Every **Book** instance has different attribute *values*!

- In Python, we typically **declare** and **initialize** attributes in a special function known as the **constructor**
- The constructor has a special name — `__init__` — and is typically defined at the top of the class before all other method definitions

# Constructing objects with `__init__`

```
class Book:
    """This class represents a book"""

    def __init__(self)
        self.author = ""
        self.title = ""
        self.year = 0
```

author, title, and year are **attributes** of the Book class

- Currently the constructor just initializes the attributes to some default values
- Ideally, the constructor should take inputs just like any other function in order to initialize the attributes to the desired values

# A Modification To Our Previous Constructor

```
class Book:
    """This class represents a book"""
    # declare Book attributes
    def __init__(self, book_author, book_title, book_year)
        self.author = book_author
        self.title = book_title
        self.year = book_year
```

The arguments are the same as arguments for a function.

# Constructing objects with `__init__`

```
class Book:
    """This class represents a book"""


    def __init__(self, book_author, book_title, book_year)

        """ The constructor """
        self.author = book_author

        self.title = book_title

        self.year = book_year
```

```
>>> lotr = Book("J.R.R. Tolkein", "Lord of the Rings", 1954)
>>> lotr.year
1954
>>> print(lotr)
<__main__.Book object at 0x108255f30>
```

- The constructor now takes inputs, but the print function doesn't allow us to comprehend the contents of this object
- To get something more meaningful we need to define a string representation for our object

# `__str__` method for Book Class

```
class Book:
      def __init__(self, book_author, book_title, book_year)

     """ The constructor """
           self.author = book_author

           self.title = book_title

           self.year = book_year

     def to_string(self):
    """ Method that defines string
        representation of book objects
    """
           return "'"+self.title+"', by "+self.author+", in "+ str(self.year)
```

```
>>> lotr = Book("J.R.R. Tolkein", "Lord of the Rings", 1954)
>>> print(lotr.to_string())
'Lord of the Rings', by J.R.R. Tolkien, in 1954
```

But wouldn't this be more convenient if we could just print(lotr) or str(lotr)? (foreshadowing!!)

# Next up

- So we have attributes and methods
- What actually happens when we create a new instance of a class?
- More on this (and more!) next time