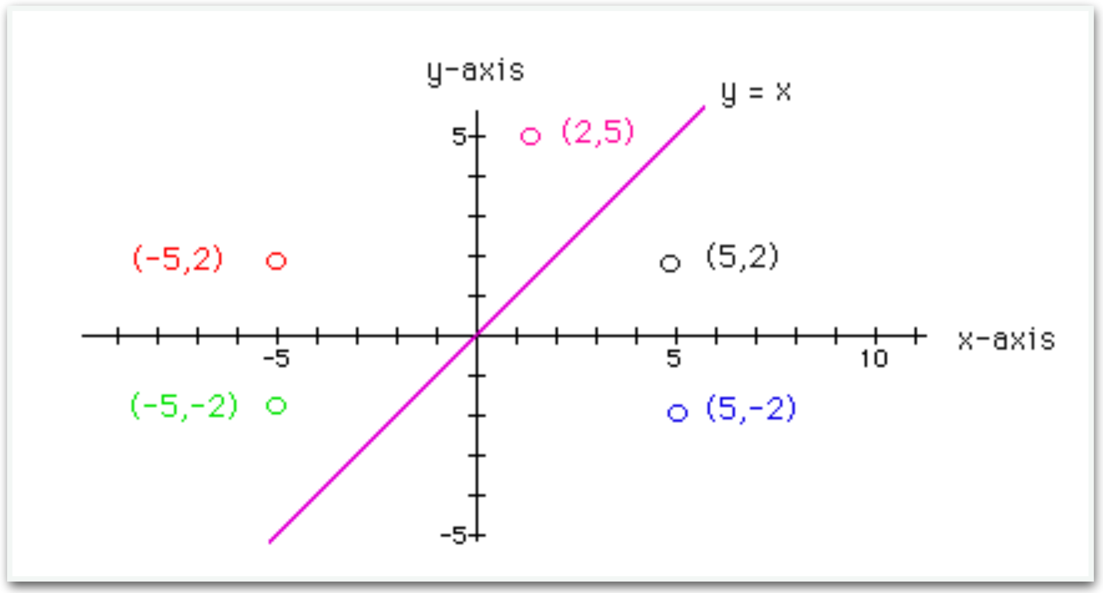
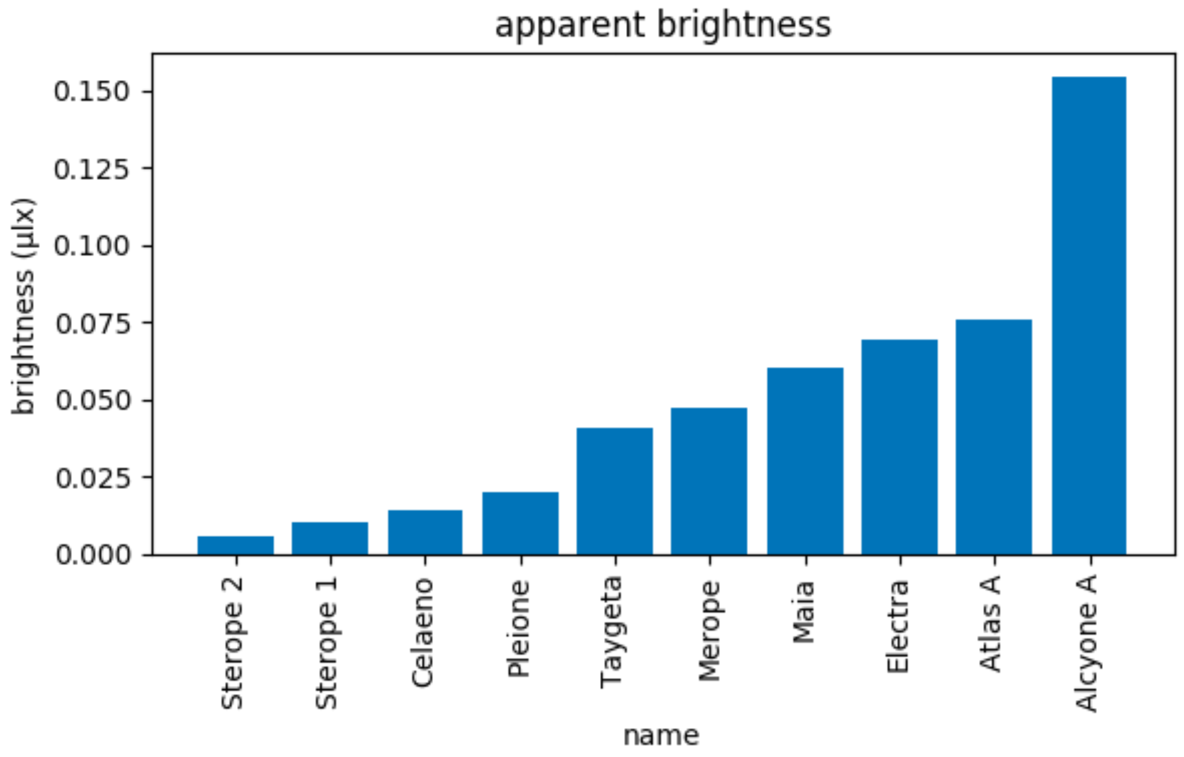
Classes III

Lecture Outline

- Special Methods and Operator Overloading
  - Overloading operators for the Coordinate class
- Example of user-defined types and why its the right approach
  - Stars lab using classes

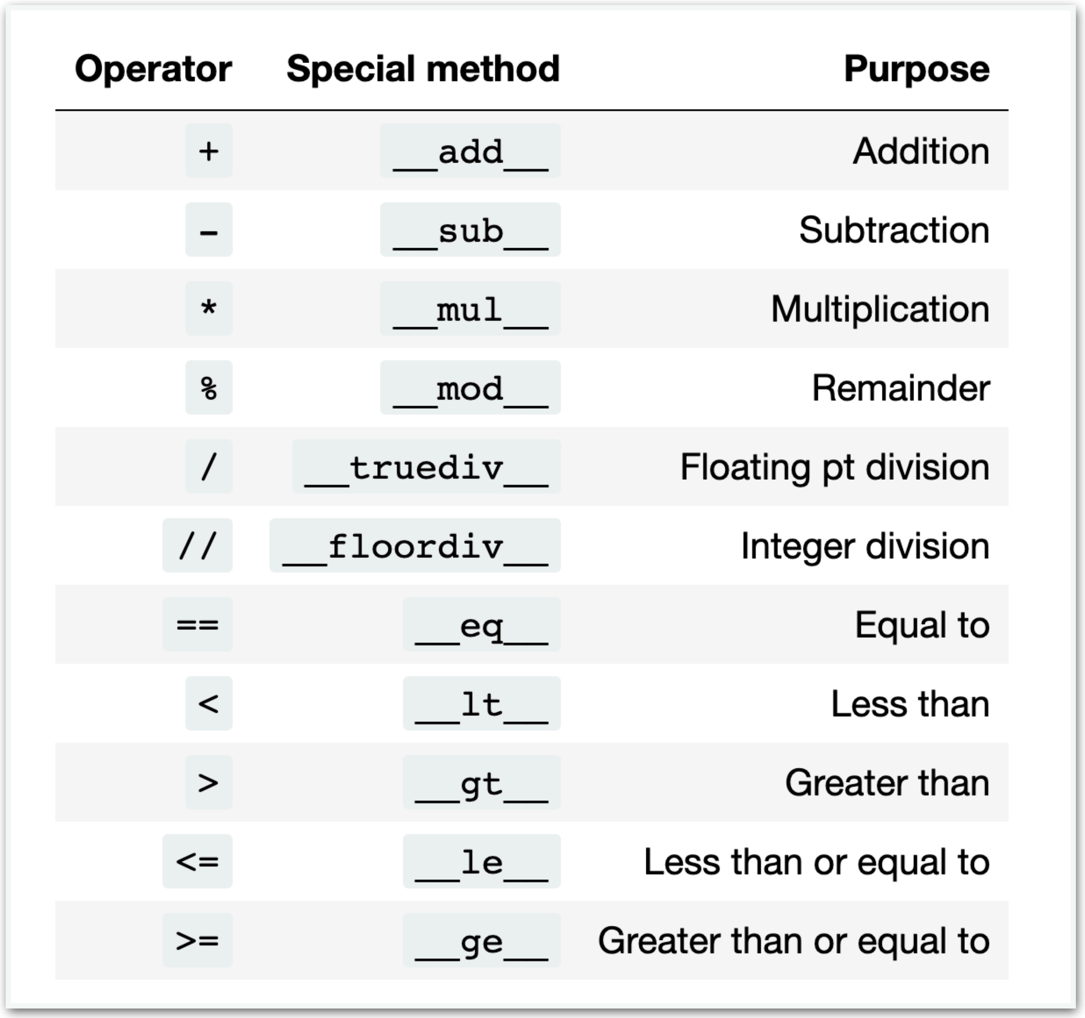




Special Methods and Operator Overloading

Operator Overloading

- **Special methods.** Method names starting and ending with __ such as `__init__` and `__str__` are special. Python has other special methods that, whenever it is appropriate, we can "customize."
- When we are changing an operator or methods default behavior, we say we are **"overloading"** it.



Operator	Special method	Purpose
`+`	`__add__`	Addition
`-`	`__sub__`	Subtraction
`*`	`__mul__`	Multiplication
`%`	`__mod__`	Remainder
`/`	`__truediv__`	Floating pt division
`//`	`__floordiv__`	Integer division
`==`	`__eq__`	Equal to
`<`	`__lt__`	Less than
`>`	`__gt__`	Greater than
`<=`	`__le__`	Less than or equal to
`>=`	`__ge__`	Greater than or equal to

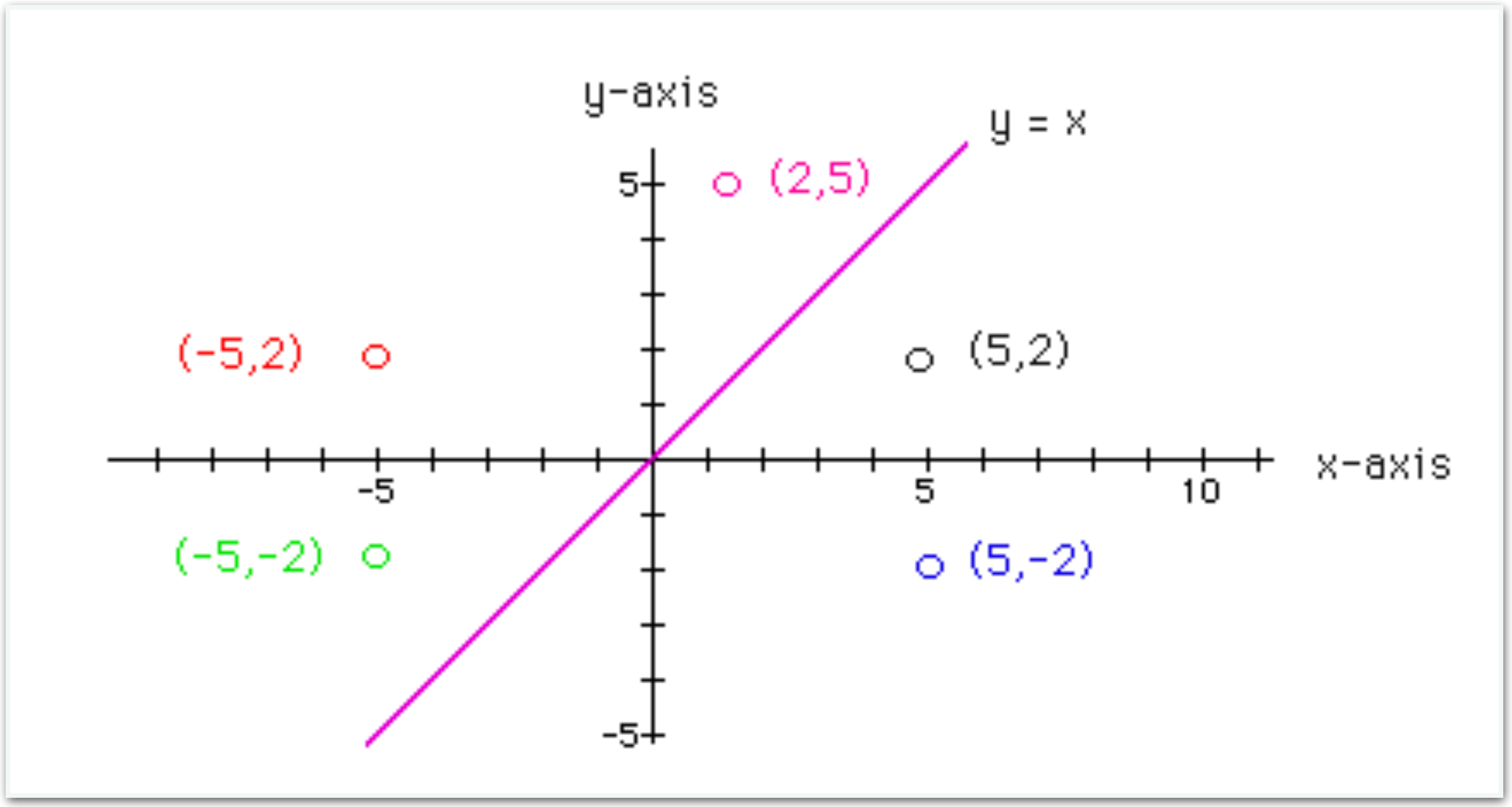
Example: Name Class

- Defining a Name class, and overloading '=', '<' and '>'
- Two names are the same if they have the same first and last names in lower case
- One name is less than another if the first letter of last name is appears earlier in alphabetical order
- Greater than is opposite of less than

In [1]:

```
class Name(object): # optional parent class
    """Class to represent a person's name."""
    __slots__ = ['_f', '_m', '_l']

    def __init__(self, first, last, middle=''):
        self._f = first
        self._m = middle
        self._l = last

    def __eq__(self, other): # both first, last name same in lower case
        return (self._f.lower() == other._f.lower()) and (self._l.lower() == other._l.lower())

    def __lt__(self, other): # compare first letter of last name in lower case
        return (self._l[0].lower() < other._l[0].lower())

    def __gt__(self, other): # compare first letter of last name in lower case
        return not self.__lt__(other)
```

Operator Overloading in `Coordinate` Class



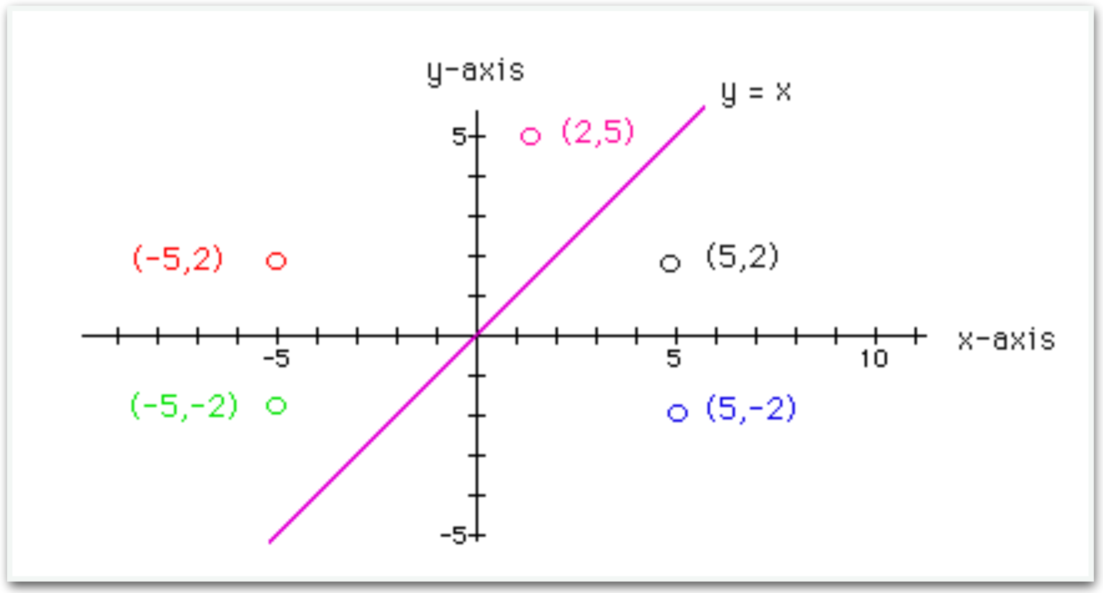
Star Class

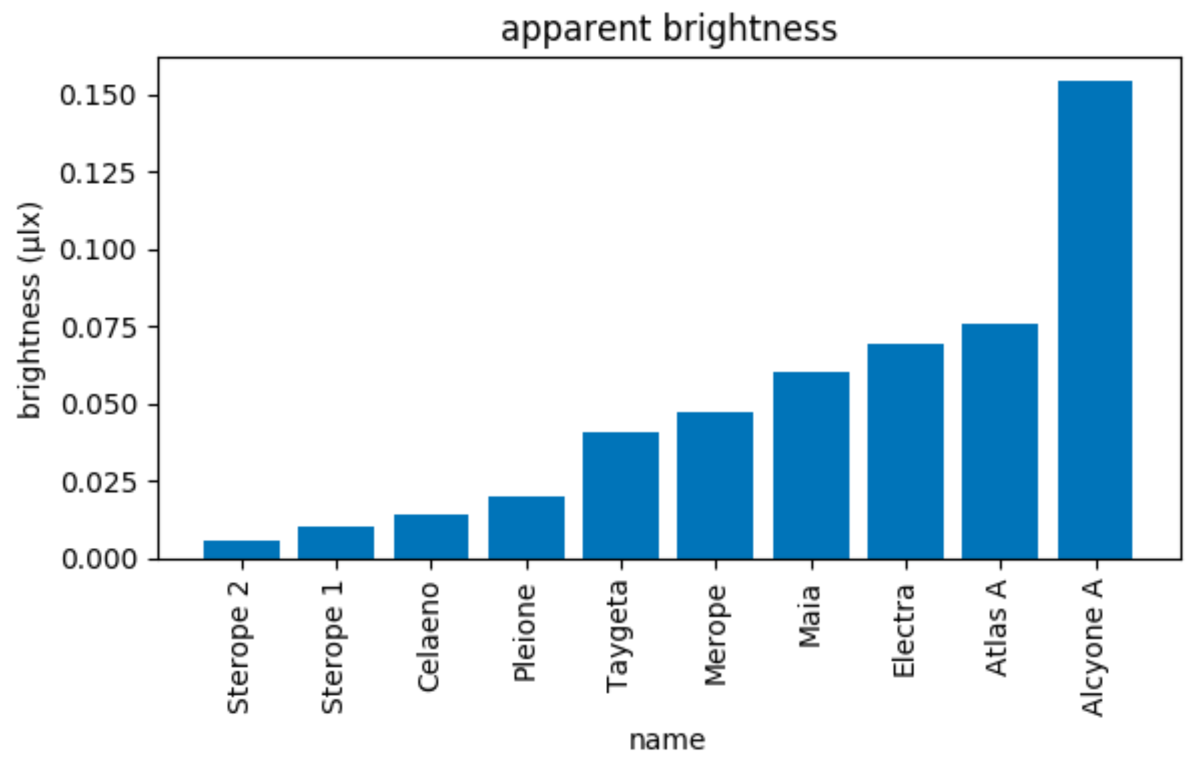
Why Define Own Type vs Dict

- In lab 5, we used an **in-built type** (dictionary) for storing the star data (name, brightness, distance)
- This approach has several downsides
  - Each star has only three attributes known a-priori ,
    - A dictionary, which is a mutable variable-size data structure is not ideal for this
  - Access to star data should be private, users (who are plotting) should not be accidentally modify it
    - Giving user access to the dictionary which is mutable is not safe
    - `@property` annotation gives only read-only access to the star name/brightness, users cannot use it to modify the attribute

Summary

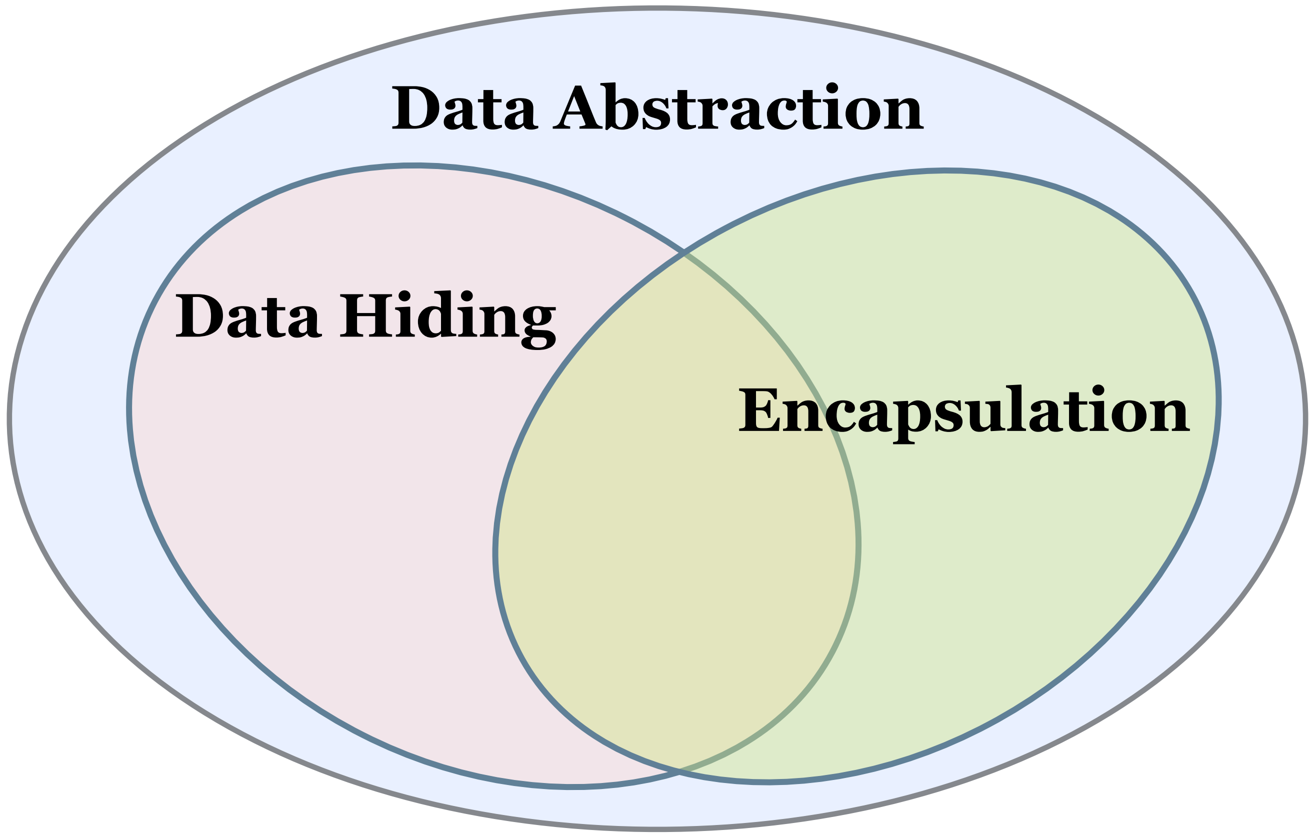
- Implementing special methods corresponding to arithmetic and logical operators lets us tailor how they work when applied to our user-defined objects
- Defining our own type has many benefits over using a pre-defined types:
    - greater control over access and functionality
    - cleaner, modular code





Data Abstraction

- We will learn about how Python supports **data abstraction** (separating the data and details of the implementation from the user) via :
  - **Data hiding:** via attribute naming conventions (private, public)
  - **Encapsulation:** bundling together of data and methods that provide an interface to the data



Acknowledgments

These slides have been adapted from:

- http://cs111.wellesley.edu/spring19 and
- https://ocw.mit.edu/courses/electrical-engineering-and-computer-science/6-0001-introduction-to-computer-science-and-programming-in-python-fall-2016/
- https://www.python-course.eu/python3_object_oriented_programming.php