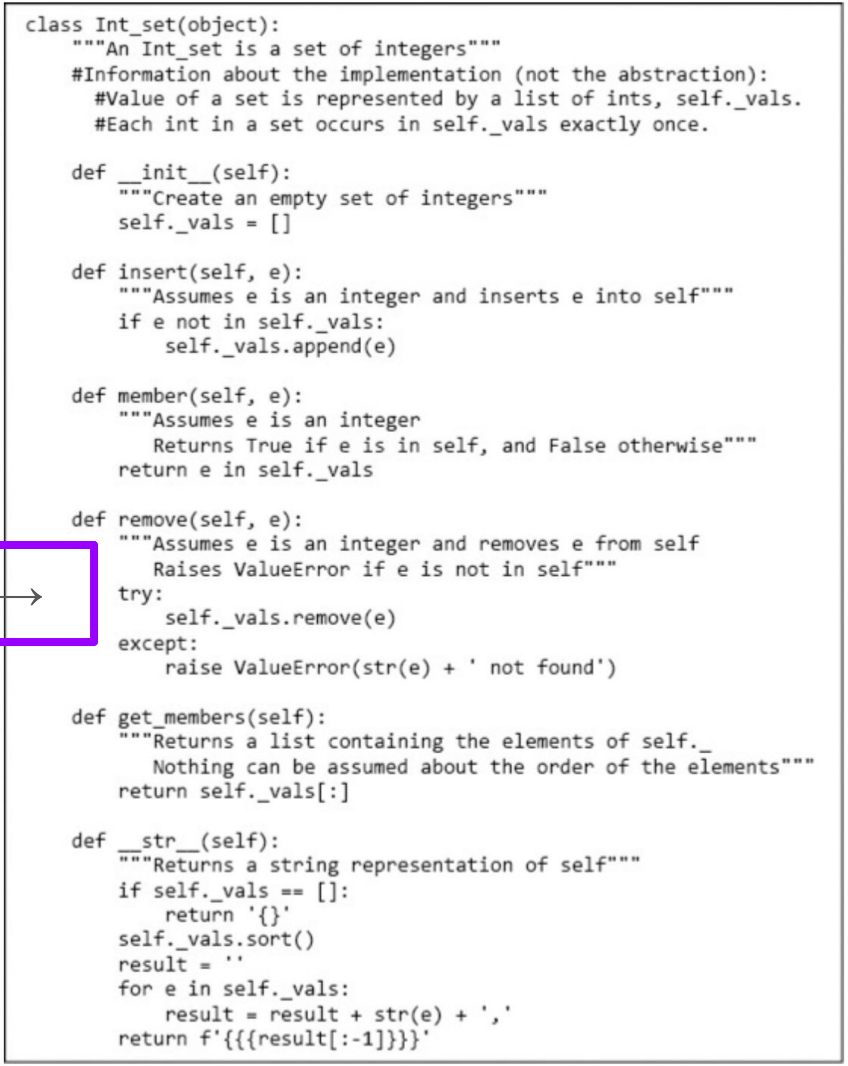
OOP continued

Polymorphism, Inheritance, Encapsulation

Book example



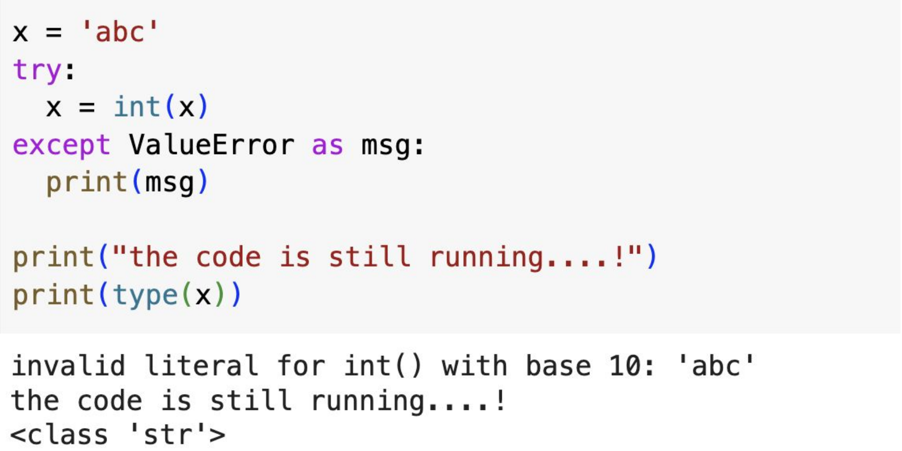
```
class Int_set(object):
    """An Int_set is a set of integers"""
    #Information about the implementation (not the abstraction):
      #Value of a set is represented by a list of ints, self._vals.
      #Each int in a set occurs in self._vals exactly once.

    def __init__(self):
        """Create an empty set of integers"""
        self._vals = []

    def insert(self, e):
        """Assumes e is an integer and inserts e into self"""
        if e not in self._vals:
            self._vals.append(e)

    def member(self, e):
        """Assumes e is an integer
           Returns True if e is in self, and False otherwise"""
        return e in self._vals

    def remove(self, e):
        """Assumes e is an integer and removes e from self
           Raises ValueError if e is not in self"""
        try:
            self._vals.remove(e)
        except:
            raise ValueError(str(e) + ' not found')

    def get_members(self):
        """Returns a list containing the elements of self._
           Nothing can be assumed about the order of the elements"""
        return self._vals[:]

    def __str__(self):
        """Returns a string representation of self"""
        if self._vals == []:
            return '{}'
        self._vals.sort()
        result = ''
        for e in self._vals:
            result = result + str(e) + ','
        return f'{{{result[:-1]}}}'
```

Exception Handling →

Aside… exception handling

```
try:
    # a line of code which may or may not work
except SomeError:
    # code for what to do if there is a SomeError
else:
    # other code
```

- remember, the point of exception handling is that the code does not crash!



```python
x = 'abc'
try:
  x = int(x)
except ValueError as msg:
  print(msg)

print("the code is still running....!")
print(type(x))
```

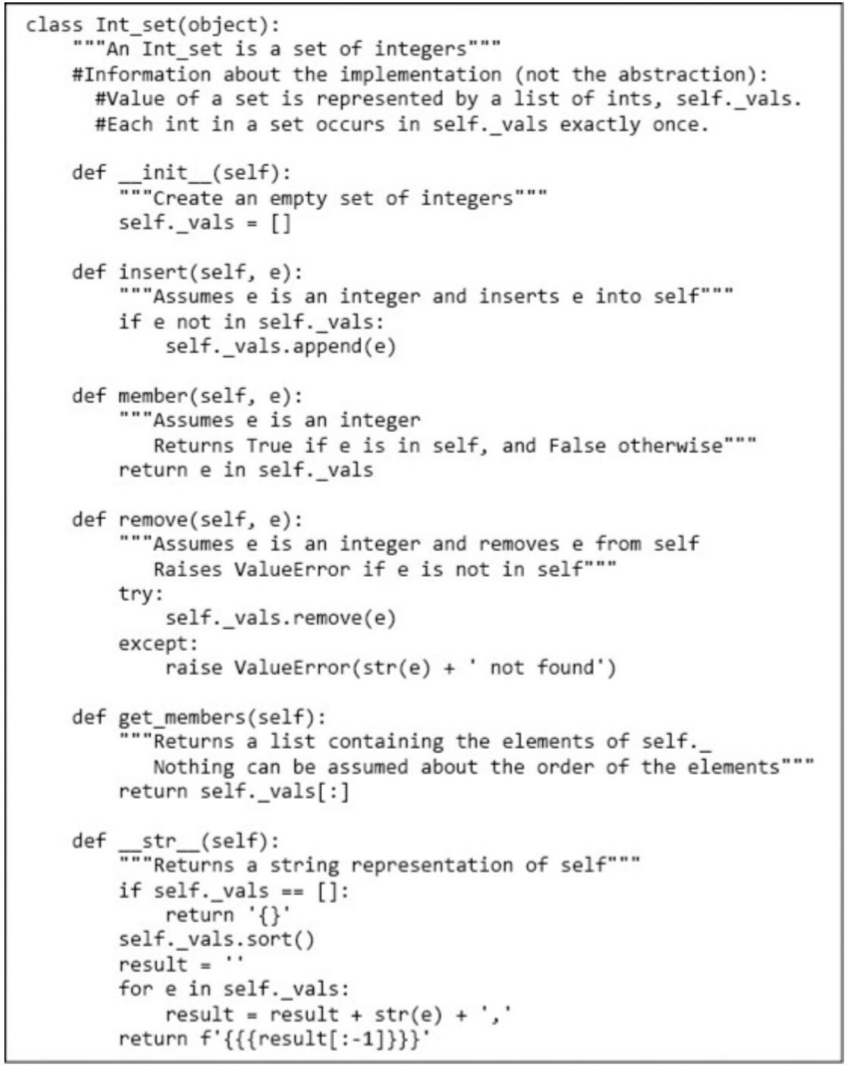
```
invalid literal for int() with base 10: 'abc'
the code is still running....!
<class 'str'>
```

What are classes for?

What does this look like?

Could you use a class to implement a special kind of set that doesn't allow even numbers?

Could you use a class to implement a dictionary?



```
class Int_set(object):
    """An Int_set is a set of integers"""
    #Information about the implementation (not the abstraction):
      #Value of a set is represented by a list of ints, self._vals.
      #Each int in a set occurs in self._vals exactly once.

    def __init__(self):
        """Create an empty set of integers"""
        self._vals = []

    def insert(self, e):
        """Assumes e is an integer and inserts e into self"""
        if e not in self._vals:
            self._vals.append(e)

    def member(self, e):
        """Assumes e is an integer
           Returns True if e is in self, and False otherwise"""
        return e in self._vals

    def remove(self, e):
        """Assumes e is an integer and removes e from self
           Raises ValueError if e is not in self"""
        try:
            self._vals.remove(e)
        except:
            raise ValueError(str(e) + ' not found')

    def get_members(self):
        """Returns a list containing the elements of self._
           Nothing can be assumed about the order of the elements"""
        return self._vals[:]

    def __str__(self):
        """Returns a string representation of self"""
        if self._vals == []:
            return '{}'
        self._vals.sort()
        result = ''
        for e in self._vals:
            result = result + str(e) + ','
        return f'{{{result[:-1]}}}'
```

Think about…

What you need the object to do?

- what kind of information will be stored
- how can someone **set** or **get** the information?
- what methods are needed to keep the information consistent and organized within the object?

What might someone do with the object?

- Will they store the object in a set?
- Will they store the object as a dict key?

Magic methods make classes/objects more usable

"One of the design goals for Python was to allow programmers to use classes to define new types that are as easy to use as the built-in types of Python."

- what is needed for use in a set?
- what is needed for use as a dict key?

__hash__(self)

- a hash is a unique "id" number that is associated with an object
- if an object is hashable, it can be stored in a **set** or **dictionary**!
- default hash value is derived from the object's identity
- fine print (^^^ only if there is no __eq__(self, other) defined)

__add__(self, other)

- if + is implemented, user can use concepts like concatenation interchangeably with other types that also use + for concatenation
  - i.e. tuples, lists, strings

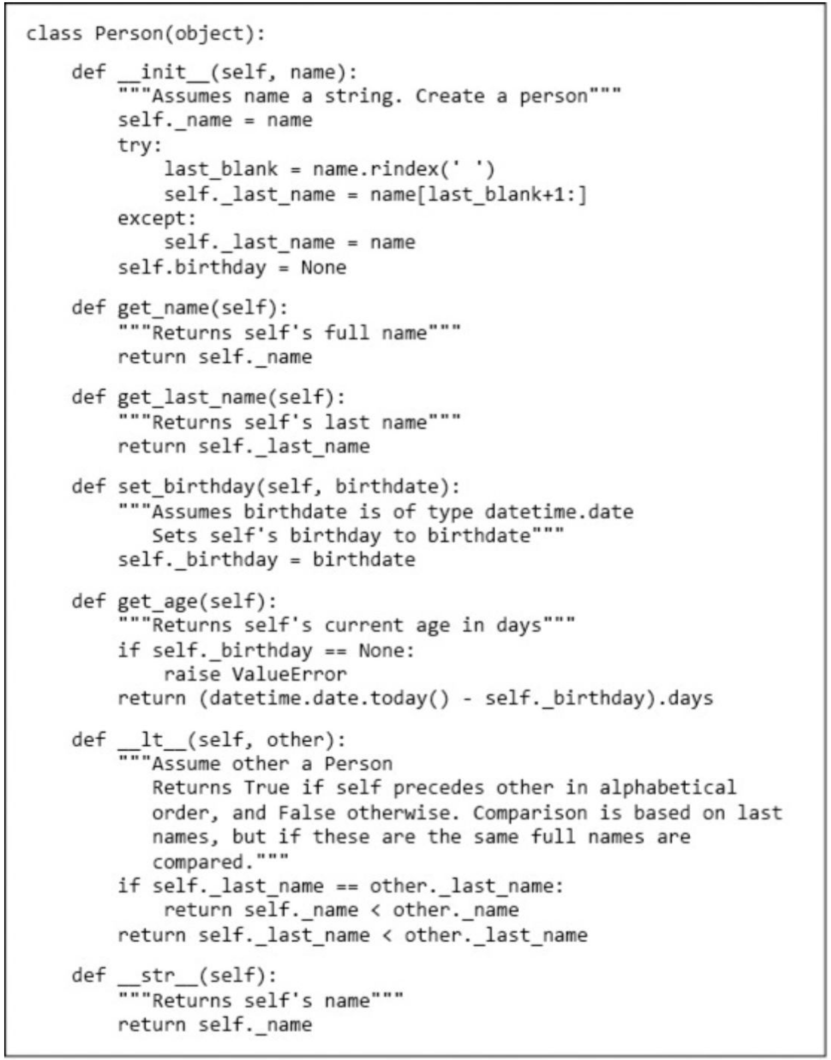
Polymorphism

- concept of adaptability !
  - "abc" + "def" = "abcdef"
  - 5 + 3 = 8 (n.b. it does not equal 53)
  - the + is polymorphic
  - for the abstract data types of str and int, + has been implemented using __add__(self, other)
  - "overloaded"

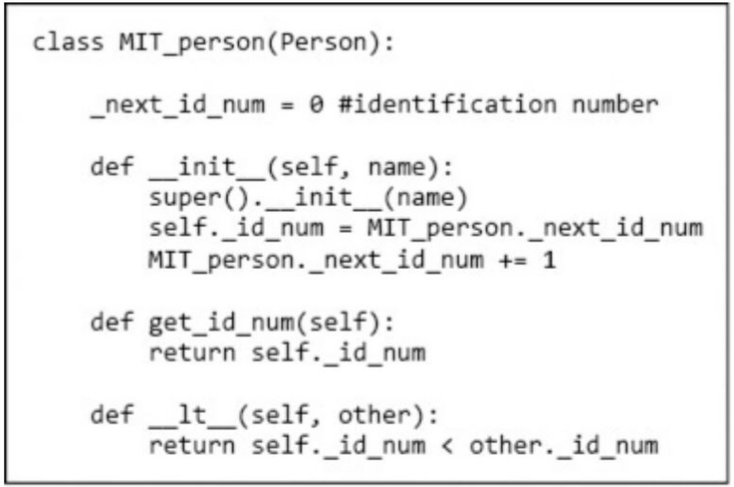
Inheritance

"Inheritance provides a convenient mechanism for building groups of related abstractions. It allows programmers to create a type hierarchy in which each type inherits attributes from the types above it in the hierarchy."

- a subclass inherits the attributes and methods of its superclass
- a subclass **"IS A"** superclass
- subclass attributes and methods are given preference if implemented
- subclass implementations **override** the superclass implementations



```
class Person(object):

    def __init__(self, name):
        """Assumes name a string. Create a person"""
        self._name = name
        try:
            last_blank = name.rindex(' ')
            self._last_name = name[last_blank+1:]
        except:
            self._last_name = name
        self.birthday = None

    def get_name(self):
        """Returns self's full name"""
        return self._name

    def get_last_name(self):
        """Returns self's last name"""
        return self._last_name

    def set_birthday(self, birthdate):
        """Assumes birthdate is of type datetime.date
           Sets self's birthday to birthdate"""
        self._birthday = birthdate

    def get_age(self):
        """Returns self's current age in days"""
        if self._birthday == None:
            raise ValueError
        return (datetime.date.today() - self._birthday).days

    def __lt__(self, other):
        """Assume other a Person
           Returns True if self precedes other in alphabetical
           order, and False otherwise. Comparison is based on last
           names, but if these are the same full names are
           compared."""
        if self._last_name == other._last_name:
            return self._name < other._name
        return self._last_name < other._last_name

    def __str__(self):
        """Returns self's name"""
        return self._name
```



```
class MIT_person(Person):

    _next_id_num = 0 #identification number

    def __init__(self, name):
        super().__init__(name)
        self._id_num = MIT_person._next_id_num
        MIT_person._next_id_num += 1

    def get_id_num(self):
        return self._id_num

    def __lt__(self, other):
        return self._id_num < other._id_num
```

- Which one is superclass?
- what methods have been overridden?
- what methods are inherited?

Warning: "Sometimes, the subclass overrides methods from the superclass, but this must be done with care. In particular, important behaviors of the supertype must be supported by each of its subtypes. "

Encapsulation

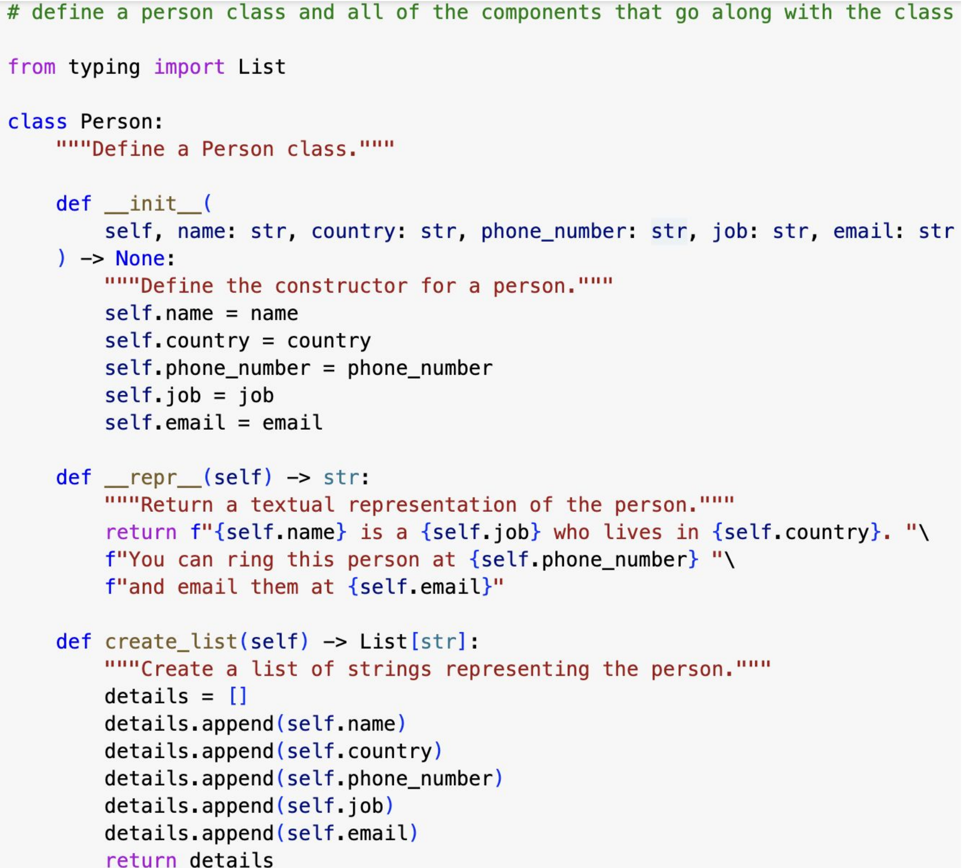
"the bundling together of data attributes and the methods for operating on them"

- Things that are related are together in one class
  - person: methods getting and setting name, attribute that stores the name data!
- Appropriate information hiding
  - the outside world will see getters and setters, but not actually mess with the data
  - classes use _ (or __) to designate private attributes and methods that should not be accessed outside of the class.

Back to basic example

Person Data Management Example

see colab



```
# define a person class and all of the components that go along with the class

from typing import List

class Person:
    """Define a Person class."""

    def __init__(
        self, name: str, country: str, phone_number: str, job: str, email: str
    ) -> None:
        """Define the constructor for a person."""
        self.name = name
        self.country = country
        self.phone_number = phone_number
        self.job = job
        self.email = email

    def __repr__(self) -> str:
        """Return a textual representation of the person."""
        return f"{self.name} is a {self.job} who lives in {self.country}. "\
        f"You can ring this person at {self.phone_number} "\
        f"and email them at {self.email}"

    def create_list(self) -> List[str]:
        """Create a list of strings representing the person."""
        details = []
        details.append(self.name)
        details.append(self.country)
        details.append(self.phone_number)
        details.append(self.job)
        details.append(self.email)
        return details
```