Python Classes, "data struct"s and Files

Intro to OOP - what's the buzz?

There is a lot of history and opinions (go and read Robert C. Martin's "Clean Architecture"). But we will keep it simplified for our purposes.

- So far we used **procedural**- programming: thinking mainly about the procedures - functions - needed for our computation.
- In Object Oriented Programming we are thinking about the **entities** - objects - in our problem domain.

Lets talk about a the recipe example. **Actions** vs **Ingredients**

- One is not better than the other, they are just different tools that might be more suitable for different problems.
    - Some say that OOP is easier to understand and maintain.
- Focus on the operator, and not the operations.

What we left out

A lot of important things

- Polymorphism
- Inheritance
    - interface or implementation
- `static` and `class` methods.
- encapsulation - which is **really** important
    - (so we are going to talk about it)
- so much more...

Encapsulation: Hiding implementation

- When My kid tells me "Please call Mommy" she does not care how I do it.
- The car example:
  - A clear **interface** that you learn when you get you'r driver's license
  - Only mechanics know the *implementation*

```
- It's different for different cars.
```

Encapsulation: Hiding internal state

- A newborn baby has (roughly) 5 internal state:
  - OK
  - hungry
  - tired
  - uncomfortable
  - in pain

- But it has an interface of 4 **mutation** methods:
  - feed(food)
  - put to sleep()
  - change diaper(new_diaper)
  - take to doctor(doctor)
- and only 2 *getter* methods:
  - is_crying()
  - is_sleeping()

BAD INTERFACE DESIGN. To understand the internal state I need to try and use the mutations and see the reaction in the getters.

The (soft) definition we are going to use:

An object is

a combination of **BEHAVIOR/ACTIONS**

and (usually internal, encapsulated) **STATE**

OOP objects vs data structures (records)

Data structures (`structs` in *C*, `record` in *pascal*) **ARE NOT** OOP objects. They are a collection of related data.

- They don't have a behavior
    - they do not operate (not active), they are operated upon (passive)
- The data is open and public, not encapsulated

So: no behavior and no internal state

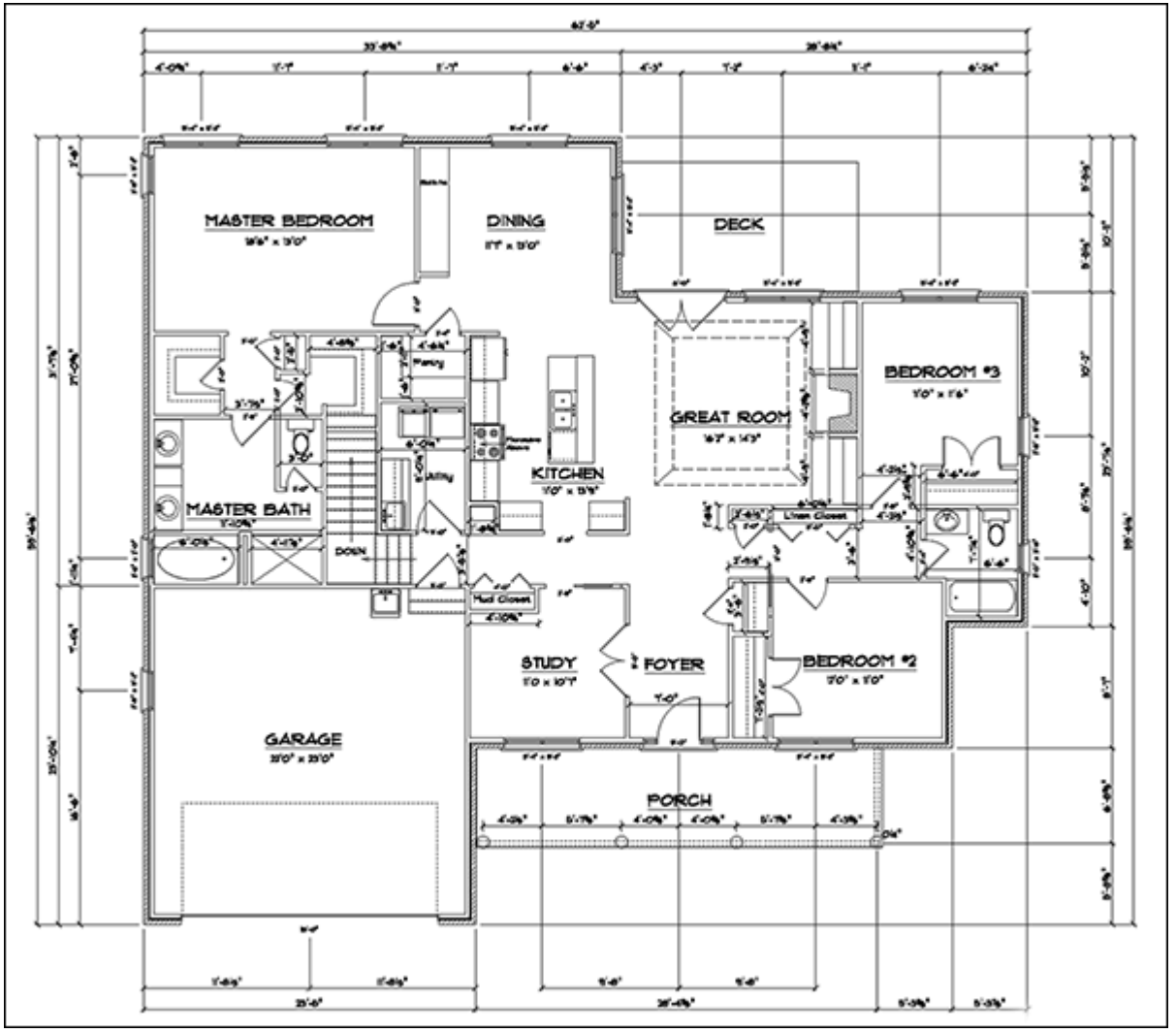
Example: A car vs my ID card

- My ID card has data: name, birth date, gender, ALIYAH date
- My car has
    - **Actions** (behavior, operations): accelerate, break, steer, shift-gear
    - Internal **state** - affected by the operations: is_running, speed, RPM, gear, gas levels, etc...

note that for my car - some of that internal state can be accessed and viewed, but I can't control the whole internal state directly

Something to think about

All cars have pretty much the same *interface* (accelerate, break, steer, etc...) but different car types use different *implementations* (acceleration on an electric car is very different from a V8 engin)


MASTER BEDROOM
DINING
DECK
GREAT ROOM
BEDROOM #3
KITCHEN
MASTER BATH
STUDY
FOYER
BEDROOM #2
GARAGE
PORCH

Classes: a blue-print for objects

In most normal OOP languages (unlike JS) we use a `class` as the blue-print for the objects. The class defines the object. A class

- Describes interfaces (what operations are allowed) - always
- Describes implementations - sometimes
    - Describes how a new object is created - a constructor - this is a specific implementation
- Describes internal state - sometimes.

Those *sometimes* are out of scope for this lesson, so for our purpose, lets say *always*.

Think about a car - a driver does not care about the engine (until it breaks - need to debug it) but the TOYOTA **factory** defines all the inner workings and wiring, and also the process of creating a car.

Done with the theory.

Questions so far?

In Python - video game example

```
class MonsterInGame:    # class definition

  # constructor
   def __init__(self, img: pathlib.Path, health: int, damage:int, speed: int):
    self._img = img           # these are fields/properties/attributes
    self._health = health
    self._damage = damage
    self._speed = speed
    self._position = (0,0)    # tuple

  def move(self, direction: Tuple[int, int]):  # method
    self._position = (
      self._position[0] + direction[0] * self._speed,
      self._position[1] + direction[1] * self._speed
    )

  def position(self) -> Tuple[int, int]:  # getter method
    return self._position

  def take_damage(self, damage: int):  # sorry for cramping the methods
    self._health -= damage

  def is_alive(self):
      return self._health > 0

  def damage(self):
      return self._damage
```

How to use classes?

```
import random

from characters import Player, MonsterInGame

player = Player()   # a class with an empty constructor - not args required
monsters = []
for i in range(20):
  m = MonsterInGame(
    img = random.choice(["goblin.png", "orc.png", "troll.png"]),
    health = random.randint(40, 100),
    damage = random.randint(4, 10),
    speed = random.randint(1, 3)
  )
  monsters.append(m)
```

Somewhere down the line:

```
for monster in monsters:
  if monster.position() == player.position():
    monster.take_damage(player.damage())
    player.take_damage(m.damage())

# List comprehension. You don't have to understand it right now.
monsters = [m for m in monsters if m.is_alive()]
```

Something to think about #2

Did you notice that the player object and the monster objects have some things in common? This makes sense, and also, interesting.

Some naming conventions

- **Classes names** use (in most languages) `CamelCase`
  - In python **variable names** use `snake_case`
- Python does not have the concept of **private** (that is, internal) methods or fields everything is (technically) public.
  - But you can use an underscore (_) prefix to signal other developers and the IDE that this should be treated as a private

```
class ToyotaCorolla:

  def __init__(self, max_speed):
      self.max_speed = max_speed        # "public" field
      self._gear = "D"  # "private" field - not in completions

  def shift_to_reverse(self, added_speed):
    if self._check_rear_sensor():
      self.beep()
    self._gear = "R"

  def _check_rear_sensor(self) -> bool:   # "private" method - not in completions
      someone_behind = do_something()
      return someone_behind
```

Any Questions?

Lets look at some familiar code

```
words_with_s = []
for i in range(10):
  s = input("Please enter a word")
  if s.uppper().count("S") >= 1:
    words_with_s.append(s)

while len(words_with_s) > 0:
  print(words_with_s.pop())
```

- So now you see, `str` and `list` (and `dict`) are all objects
  - with *shortcut* constructors.

A function to add to (all of) your `classes`

```
class MySpecialClass:

  def __init__(self, some_args.....)
      pass # something

  def __repr__(self) -> str:
      return "A string for debugging"

  def __str__(self) -> str:
    return "A string for debugging 2"
```

These are the functions that are called when you use `print()`

A moment to talk about `sets`

- An **Unordered** collection of **unique** elements.
  - Unique meaning no repetition.

```
# DEFINITIONS
my_set = set([1,2,3,4,5,6,7,8,9,10])
my_other_set = {1,3,5} # don't confuse with dict definitions
empty_set = set() # you can't use `{}`
```

```
  # MUTATIONS
  my_set.add(5) # does nothing
  my_set.add(11) # adds 11 to the set
  my_set.remove(5) # removes 5 from the set
  my_set.remove(13) # raises an error
```

```
  # TESTING
  my_set.has(4) # returns True
  4 in my_set # returns True
```

```
  # Using FOR
  for i in my_set: # REMEMBER - the order is not guaranteed.
      print(i)
```

Any Questions?

And now

for something completely different...

Working with **Files**

Files: some theoretical background

- Files are Operating System (OS) resources: Managed by the OS
  - You'r access, may prevent other programs from accessing the resource
  - You need to tell the OS when you are done with the resource
    - Sort of...
- You can look at the file as text (string) or as binary (byte stream)
- There are 3 basic modes to open a file:
  - read (`r`)
  - write (`w`) - overwrites the file if exists
  - append (`a`) - adds to the end of the file, create if does'nt exist
- and 2 more advanced modes (which we won't talk about)
- an open file has a cursor (or a position).
  - operations depend on the position
  - operation may change the position

Files in python

asking for the resource - creating the file objet with `open()`:

```
f = open("path/to/file.txt")  # default is read mode
f = open("path/to/file.txt", "w")  # write mode
f = open("path/to/file.txt", "a")  # append mode
```

More advanced stuff:

```
f = open("path/to/file.txt", "r+")  # read and write mode
f = open("path/to/file.txt", "rb")  # exclusive creation mode
```

and now I can work with this:

```
content = f.read() # read the whole file
f.write("new content") # write to the file
f.seek(0) # go back to the beginning of the file
```

There are more functions. Read about them.

Some important remarks

What about closing (releasing) the file?

- `f.close()`
- Most of the time, python's garbage collector will do it for you, but don't trust it!

OS resource operations may fail

- `open()` and each other operation (read\write) can throw an error
- Check for return values

`pathlib` to the rescue

```
import pathlib
p = pathlib.Path("path\\to\\file.txt")
content = p.read_text()
```

One of the most useful file formats:

JSON

JSON format and Files

Text based format to transfer data between systems. Based on JavaScript.

- Basic types include strings, numbers, `true`, `false` and `null`
  - Note the case for the keywords
- complex types include arrays (lists, in `[ ]`) and objects (dicts, in `{ }`)
  - the keys in objects are strings
- recursive - you can have objects inside objects, arrays inside arrays, etc...
- No comments in json files
- No trailing commas

```
{
  "name": "John",
  "age": 30,
  "cars": ["Ford", "BMW", "Fiat"],
  "married": true
}
```

```
[
  {"name": "John", "age": 30},
  {"name": "Jane", "age": 25}
]
```

Working with JSON files in python

- Builtin `json` module with all you need
  - `json.load` and `json.dump` for files
  - `json.loads` and `json.dumps` for strings
- Arrays are lists, objects are dicts

```
import json

data = {
  "name": "John",
  "age": 30,
  "cars": ["Ford", "BMW", "Fiat"],
  "married": True
}

json_string = json.dumps(data) # you can control indentation, read the docs
path = pathlib.Path("data.json")
path.open('w').write(json_string) # better to do path.write_text(json_string)
```

```
content = path.open().read() # better to do path.read_text()
data_out = json.loads(content)
```

Our first 3rd party package:

Pydantic

Quick overview of python 3rd party packages

- Python has a huge and active communities of builders
  - Maybe you'll join them some day.
- They build packages you can use
  - Most written in python, some only wrapped in python
- Packages are uploaded to the *public PYthon Package Index*: pypi
- You have a local tool to install and manage packages on your computer: `pip`

  ```
  # in your cmd/bash
  pip install <name_of_package>
  ```

  And now you can `import` you'r package
  - Many options, including limiting versions
- A moment to talk about dependency collision, and **venv**s

Pydantic - data validation and parsing

- Defines itself as a *"Data Validation library"*
  - Which means - checks that data objects (structs, records) are of the right structures and types
  - Save you writing the validation code
- Really nice for parsing and dumping to JSON
- Declarative and nice to read
- Used in some other popular libraries (like `FastAPI`)
- Python has a builtin "equivalent" - `dataclasses` but we will not be using ig for various reasons
- Installing with pip:

```
pip install pydantic
# alternative: python -m pip install pydantic
# should be version 2.*
```

Defining pydantic model

defining a *"model"*

```
from pydantic import BaseModel

class Person(BaseModel):
  name: str
  age: int
  is_married: bool = False
  cars: List[str] = []
```

- Notice the Inheritance
  - We didn't talk about it. So lets just take it as is.
- Each field has a type hint
  - And for pydantic it's not a "hint" - it's a requirement
- Fields are under the class definition, not under any function
  - Class attributes/fields
- Fields can have default values

These are the basics. There are more options and features like extra validations (range for a number, RE for a string, etc...)

Using the model

- Using the constructor (like and any class)

```
p = Person(name="John", age=30, cars=["Ford", "BMW"])
# is_married will be default value
```

- Or straight from a dict

```
data = {"name": "John", "age": 30, "cars": ["Ford", "BMW"]}
p = Person.model_validate(data)
# from json string with Person.model_validate_json(json_string)
```

  Notice we are using the **Class** method `model_validate` and not an instance method.

- And you can dump (serialize) it easily

```
# to dict
d = p.model_dump()
# or json string
d = p.model_dump_json()
```

OOP Objects or a Dataclass?

Unless you do something special, pydantic models are just **dataclasses with validation** (not behaviour, not internal/encapsulated state).

But you can:

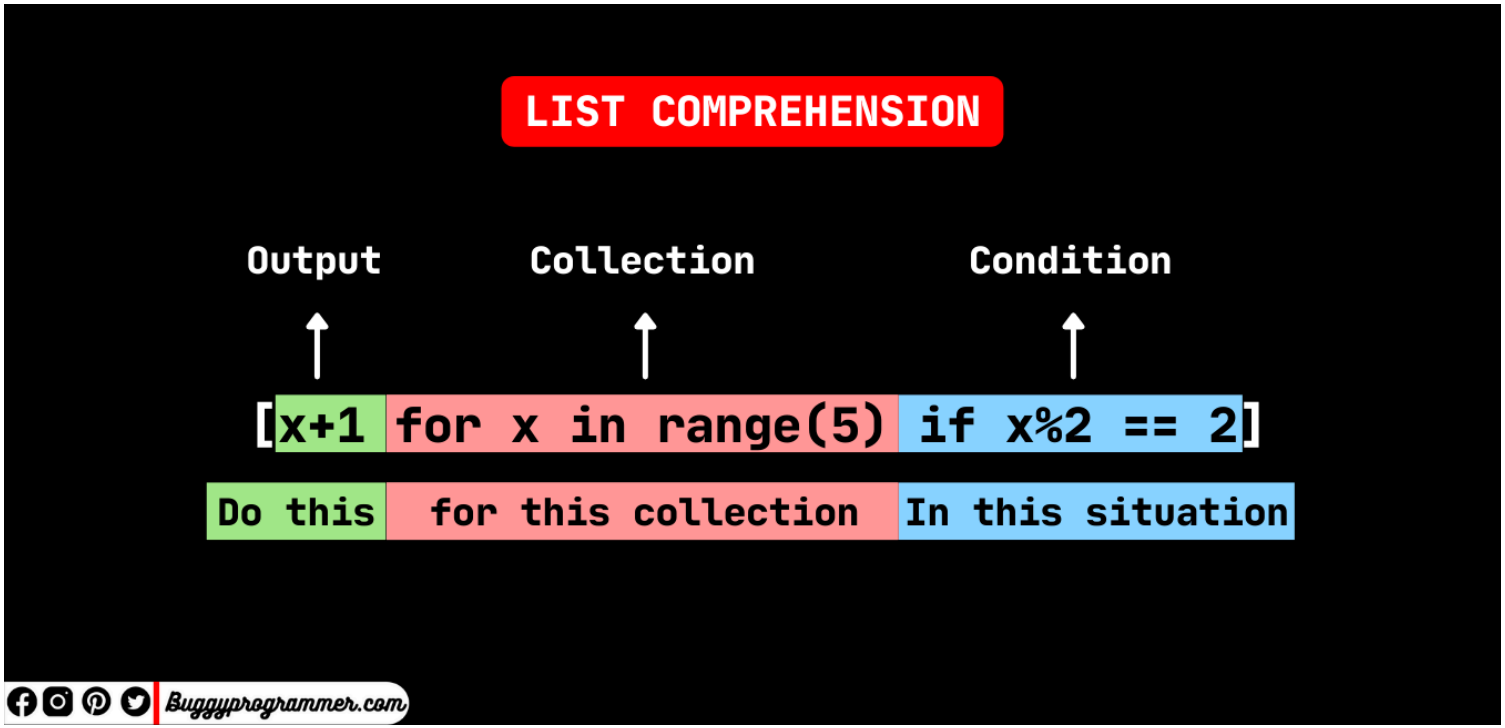
1. Add behaviour to the model (just as you would with any other class)

1. Hide the internal state (with the `Field` function)
    - But it's advanced stuff

`list` Comprehensions

- A piece of syntactic sugar to create a lists from an iterable
  - Meaning - you don't have to use this.
- It's **NOT** always the best choice
  - Sometimes a regular loop is more readable
  - Sometimes a regular loop is more efficient
- The basic idea: run a `map` and/or `filter` on an iterable
  - `map` - transform each element with some function
  - `filter` - keep only elements that pass some condition
- examples:
  - turn all the string in the list to an int
  - multiple by 2 all the int in the list
  - get only positive ints
  - get the first letter if every string that starts with a capital letter

`list` Comprehensions - how it looks



Any Questions?