Object-Oriented Programming in Python
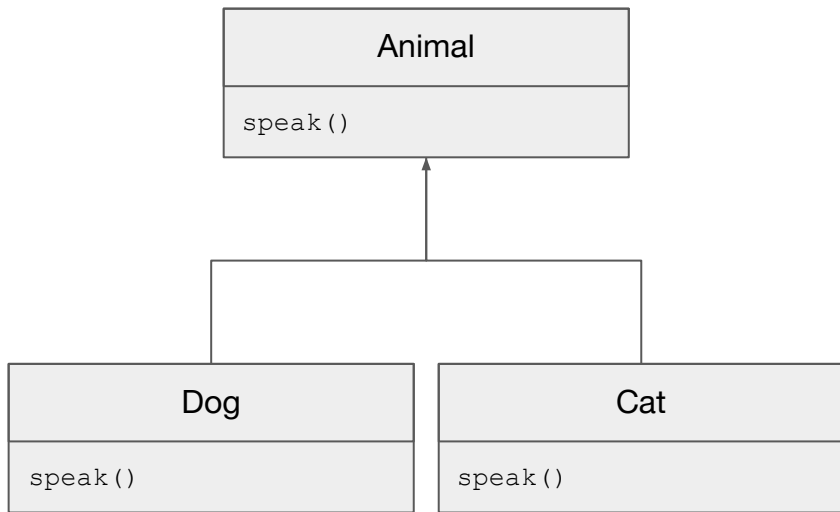
Advanced Programming & App Development (MIS 385N)
Summer 2022

Evan King

Classes and Inheritance: Animals

The usual Animal example.

- Dogs and Cats are both Animals that make sounds, but the sounds they make are different.

- `speak()` prints the subclass sound

Animal
`speak()`

Dog
`speak()`

Cat
`speak()`

Classes and Inheritance: Documents

Exercise: we are writing an application to read/write data in *different file types*.

We want modularity and flexibility:

- easily add support for new file types
- isolate programming efforts
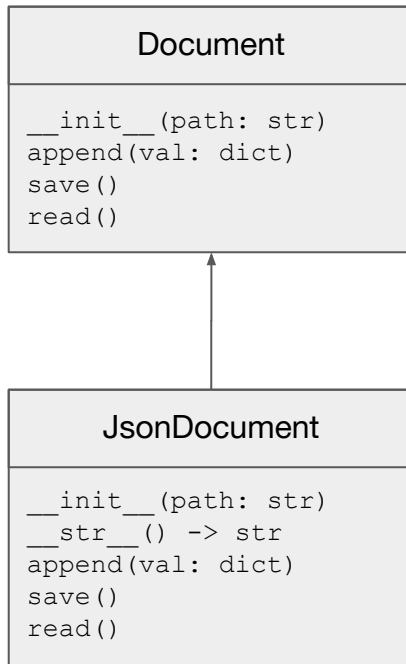- reuse old implementations in new ways

Clone today's exercises:

```
git clone https://github.com/evmaki/oopy.git
```

Document Class

Implement this class hierarchy in a
module called Document:

- **Document** superclass defines the
  high-level interface of a document

- **JsonDocument** subclass
  implements the underlying
  behavior for a specific file type

Document

`__init__(path: str)`
`append(val: dict)`
`save()`
`read()`

JsonDocument

`__init__(path: str)`
`__str__() -> str`
`append(val: dict)`
`save()`
`read()`

JsonDocument

```
def __init__(self, path: str)
```
- Opens the json file at path and stores its contents in a class attribute called doc

```
def __str__()
```
- Overrides Python's default stringification; return the underlying document as str

```
def append(self, content: dict)
```
- Appends the passed dict to the doc (assumes dict is proper structure)

```
def save()
```
- Saves doc to the json file at the original path

```
def read()
```
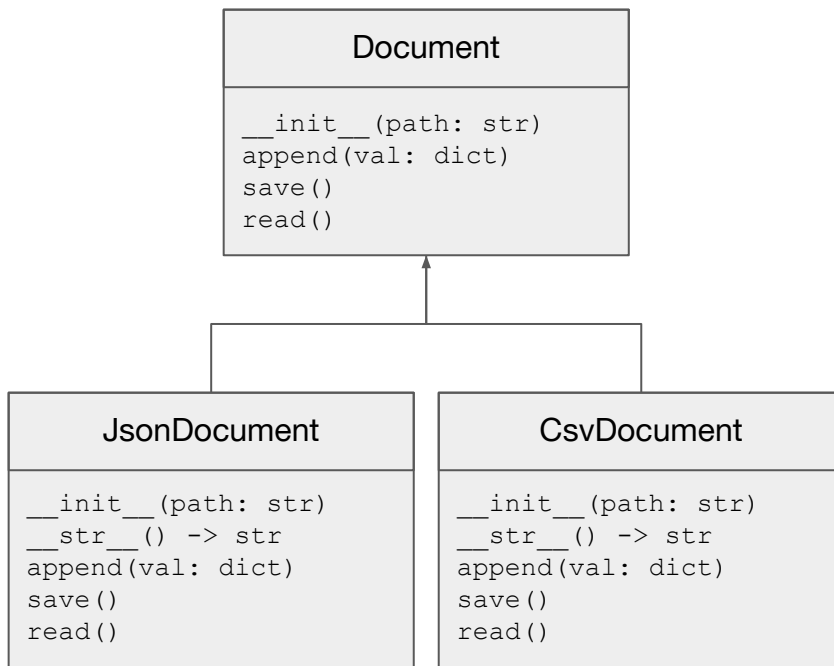- A generator which returns the contents of doc line-by-line

python's json library:

import json

New Subclass: CsvDocument

Now we want support for a new
document type: csv documents

- **CsvDocument** has all the same
  inputs and outputs as other
  Document subclasses, but the
  internal implementation differs.

```
Document
─────────────────────────
__init__(path: str)
append(val: dict)
save()
read()
```

```
JsonDocument
─────────────────────────
__init__(path: str)
__str__() -> str
append(val: dict)
save()
read()
```

```
CsvDocument
─────────────────────────
__init__(path: str)
__str__() -> str
append(val: dict)
save()
read()
```

CsvDocument

CSV files are less flexible than JSON

```
def __init__(self, path: str)
```
- Opens the csv file at path and stores its contents in a class attribute called doc

```
def __str__()
```
- Overrides Python's default stringification; return the underlying document as str

```
def append(self, content: dict)
```
- Appends the passed dict to the doc (assumes dict is proper structure)

```
def save()
```
- Saves doc to the csv file at the original path

```
def read()
```
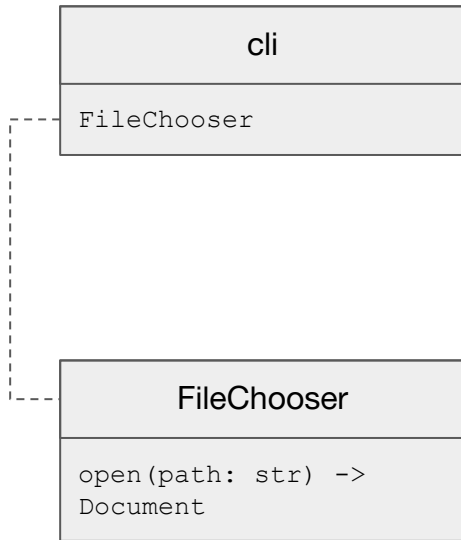- A generator which returns the contents of doc line-by-line

python's csv library:

```
import csv
```

Using the Document Class

Document is an abstraction of different file types in a standard interface. Cool.

Now we need to dynamically create those interfaces based on *user input* of *different file types*.

cli
FileChooser

FileChooser
open(path: str) -> Document

FileChooser

A "singleton" which implements one method, `open(path: str)` that takes a file path and returns the appropriate Document subclass.

The simplest singleton implementation in Python is a module that only implements functions, no classes.

Create a module named FileChooser and implement
`open(path: str) -> Document`

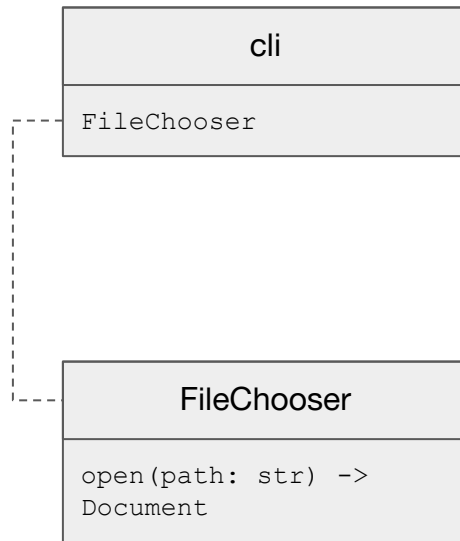| FileChooser |
| --- |
| `open(path: str) -> Document` |

A simple cli (command line interface)

We have:

- Document class which is an abstraction of several different document types
- FileChooser singleton which creates the right Document subclass based on the input

We need:

- A way to take user input

cli
`FileChooser`

FileChooser
`open(path: str) -> Document`

Implementing the cli

Implement a cli that works with the following command:

```
python cli.py [filepath] [dict string to append]
```
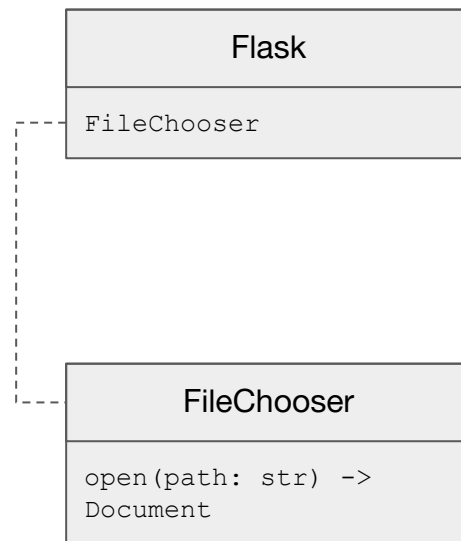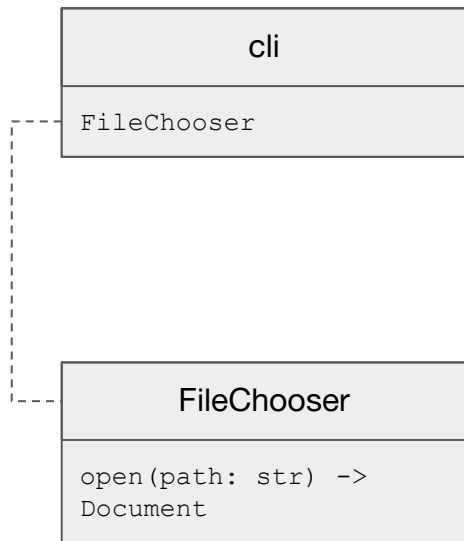
Example:

```
python cli.py ./capitals.json "{'state': 'Texas', 'capital': 'Austin'}"
```

Things you'll need:

```
import sys, ast
```

`sys.argv` - list of tokens provided on the command line

`ast.literal_eval()` - converts strings to dictionaries

cli
FileChooser

Flask
FileChooser

FileChooser
open(path: str) -> Document

FileChooser
open(path: str) -> Document

A better user interface

Now we want to use the *same* Document and FileChooser objects but through an entirely different interface – a web application.

Provided for you:

- A basic Flask `app.py`
  - Serves an `index.html` at the site root
  - Accepts HTTP POST at `/document/` endpoint
- An `index.html` file
  - Contains a form which takes input and POSTS to `/document/`

TODO:

- Implement file handling at the `/document/` route using your classes

Recap

That was a lot. What did we accomplish?

- Defined a common interface for editing Documents
- Implemented some inheriting classes to define the behavior of Documents for different file types

abstraction, inheritance

- Implemented a module, FileChooser, which instantiates the right class type at runtime

polymorphism

- Wrote a command-line interface for interacting with our new classes
- Wrote a web interface for interacting with the same classes

modularity, code reuse