CSIT121
Object-Oriented Design and Programming

Dr. Fenghui Ren

School of Computing and Information Technology
University of Wollongong

Lecture 7 outline

- How to recognize objects
- Revisit object data and behaviors
- Wrapping data behaviors using properties
- Manager object
- The don’t repeat yourself principle
- Recognizing repeated code

Identifying objects

- Identifying classes/objects is a very important task in OOA and OOP, but it isn’t always an easy job.
- Objects contain both data and behaivors.
- If you want to reuse the same set of related variables to a set of functions, consider grouping both variables and functions into a class.
- Of course, if you can complete the jobs with Python built-in data structures or functions, no need to create your own classes.
- Furthermore, there is no reason to add extra levels of abstraction if it doesn’t help organize your code.

Identifying object

A polygon class:

- A polygon can be modelled as a list of points, i.e., two tuples (x,y)

square = [(1,1), (1,2), (2,2), (2,1)]

- If we want to calculate the perimeter of the polygon, we need to sum the distances between each point.

```
def distance(p1, p2):
    return math.sqrt((p1[0]-p2[0])**2 + (p1[1]-p2[1])**2)

def perimeter(polygon):
    perimeter = 0
    points = polygon + [polygon[0]]
    for i in range(len(polygon)):
        perimeter += distance(points[i], points[i+1])
    return perimeter
```

```
>>> square = [(1,1), (1,2), (2,2), (2,1)]
>>> perimeter(square)
4.0
```

Identifying objects

- As the data (points) are used by two functions and both functions are related to the polygon, we can use a polygon class to encapsulate the list of points (data) and the functions (behavior).

- Do we also need to encapsulate the x and y coordinates and the distance method? Let see the differences.

Identifying objects

With the Point class

```
class Point:
    def __init__(self, x, y):
        self.x = x
        self.y = y

    def distance(self, p2):
        return math.sqrt((self.x-p2.x)**2 + (self.y-p2.y)**2)

class Polygon:
    def __init__(self):
        self.vertices = []

    def add_point(self, point):
        self.vertices.append((point))

    def perimeter(self):
        perimeter = 0
        points = self.vertices + [self.vertices[0]]
        for i in range(len(self.vertices)):
            perimeter += points[i].distance(points[i+1])
        return perimeter
```

```
>>> square = Polygon()
>>> square.add_point(Point(1,1))
>>> square.add_point(Point(1,2))
>>> square.add_point(Point(2,2))
>>> square.add_point(Point(2,1))
>>> square.perimeter()
    4.0
```

OO code vs functional version

- OO code isn’t compact, but easier to read and follow.
- The functional version needs more documentation to explain, but OO code is relatively self-documenting.
- If we write all the documentation for the functional version, it would probably be longer than the OO code.
- Of course, code length is not the only indicator of code complexity.
- We can also make the class Polygon as easy to use as the functional implementation.
- The new Polygon class can be constructed with multiple tuples, and the initializer can convert the tuples to point objects.

OO code vs functional version

```
class Polygon2(Polygon):
    def __init__(self, points=None):
        points = points if points else []
        self.vertices = []
        for point in points:
            if isinstance(point, tuple):
                point = Point(*point)
            self.vertices.append(point)
```

This initializer goes through the list and ensures that any tuples are converted to points. If the object is not a tuple, we leave it as is, assuming that it is either a Point object already, or an unknown duck-typed object that can act like a Point object.

OO code vs functional version

- If we have new functions that accept a polygon argument, such as calculating the area of the polygon or the number of vertices/edges of the polygon, the benefits of the OO code become more obvious.
- The distinction is a design decision.
- The functional version is more general to use and focuses on functionality only.
- OO code considers the data and the associated functions specific to that data.

OO code vs functional version

How to make the decision? There are many factors:

- How often the data and functions will be use?
  - One time only – using a function
  - Needs to work on the same set of data in a wide variety of ways – using the OO code
- How much interactions between data and functions?
  - Few and simple interactions – using a function
  - More and complex interactions – using objects (inheritance, polymorphism, association, composition)
- Programmers' personal experience

Adding behaviours to class data with properties

- In OOP, the separation of object behaviour and data is very important.
- Many object-oriented languages (such as Java) teach us never to access attributes directly.

```
class Color:
    def __init__(self, rgb_value, name):
        self._rgb_value = rgb_value
        self._name = name

    def set_name(self, name):
        self._name = name

    def get_name(self):
        return self._name
```

- The underscore prefix before variables tells us they are private variables.
- The get method (accessor) and the set method (mutator) provide different access to each variable

```
>>> c=Color("#ff0000", "bright red")
>>> c.get_name()
'bright red'
>>> c.set_name("red")
>>> c.get_name()
'red'
```

Adding behaviours to class data with properties

- This is not nearly as readable as the Python favors:

```
>>> class Color:
...     def __init__(self, rgb_value, name):
...         self.rgb_value = rgb_value
...         self.name = name
...
...
>>> c=Color("#ff0000", "bright red")
>>> print(c.name)
bright red
>>> c.name="red"
>>> print(c.name)
red
```

- So why would anyone insist upon the method-based syntax?
- It is because the method-based syntax is easy for the code maintenance and extension.

Adding behaviours to class data with properties

- We can easily extend the set_name() method to validate a given value is a suitable input.

```
def set_name(self, name):
    if not name:
        raise Exception("Invalid Name")
    self._name = name
```

- So the methods accessing the attributes should be always defined as public methods
- However, Python does not really have the concept of public or private members!

Adding behaviours to class data with properties

- In Python, the keyword ‘property’ is used to make methods that look like attributes.
- We can therefore write our code to use direct member access.
- This is how it works,

```
class Color2:
    def __init__(self, rgb_value, name):
        self.rgb_value = rgb_value
        self._name = name

    def _set_name(self, name):
        if not name:
            raise Exception("Invalid Name")
        self._name = name

    def _get_name(self):
        return self._name

    name = property(_get_name, _set_name)
```

Compared with class Color, Color2 class

- changes the name attribute into a private _name attribute
- adds two more private methods to get and set _name attribute
- Uses the property keyword to create a new attribute called name to replace the direct _name attribute

Color2 class can be used exactly the same way as the Python favor, and it now performs validation for inputs.

Adding behaviours to class data with properties

- In Python, the keyword ‘property’ is used to make methods that look like attributes.
- We can therefore write our code to use direct member access.
- This is how it works,

```
>>> c=Color2("#ff0000", "bright red")
>>> print(c.name)
bright red
>>> c.name="red"
>>> print(c.name)
red
>>> c.name=""
Traceback (most recent call last):
  File "<pyshell#312>", line 1, in <module>
    c.name=""
  File "<pyshell#307>", line 8, in _set_name
    raise Exception("Invalid Name")
Exception: Invalid Name
```

- property will call the two methods (set and get) to access or change the _name attribute
- Be careful, even with the property, people still can access the _name attribute directly in Python.

Properties in details

- The property function returns an object that proxies any requests to set or access the attribute value through the methods specified.
- The property built-in is like a constructor.
- The property constructor can also accept two additional arguments, i.e., a delete function and a docstring for the property.
- The delete function can be useful for logging the fact that a value has been deleted.
- The docstring is just a string describing what the property does.

Properties in details

```
class Silly:
    def _get_silly(self):
        print("You are getting silly")
        return self._silly

    def _set_silly(self, value):
        print("You are making silly {}".format(value))
        self._silly = value

    def _del_silly(self):
        print("Whoah, you killed silly!")
        del self._silly

    silly = property(_get_silly, _set_silly, _del_silly,
                     "This is a silly property")
```

```
>>> s = Silly()
>>> s.silly="funny"
You are making silly funny
>>> s.silly
You are getting silly
'funny'
>>> del s.silly
Whoah, you killed silly!
```

```
>>> help(Silly)
Help on class Silly in module __main__:

class Silly(builtins.object)
 |  Data descriptors defined here:
 |
 |  __dict__
 |      dictionary for instance variables (if defined)
 |
 |  __weakref__
 |      list of weak references to the object (if defined
)
 |
 |  silly
 |      This is a silly property
```

- In practice, properties are mostly used to define the getter and setter functions.
- The delete function is often left empty because object attributes are very rare.
- If we try to delete a property that doesn't have a delete function specified, it will raise an exception.

Decorators

Besides the property keyword, the property function can also be defined with the decorator syntax.

- @property decorator marks the silly() function as a property, it is the same as silly = property(silly)
- @silly.setter decorator marks the silly(value) function as a setter attribute of the project returned by the property function.
- @silly.delete decorator marks the delete the property
- The class SillyDecorated operates exactly the same as the Silly class.

```
class SillyDecorated:
    @property
    def silly(self):
        "This is a silly property"
        print("You are getting silly")
        return self._silly

    @silly.setter
    def silly(self, value):
        print("You are making silly {}".format(value))
        self._silly = value

    @silly.deleter
    def silly(self):
        print("Whoah, you killed silly!")
        del self._silly
```

```
>>> s = SillyDecorated()
>>> s.silly="funny"
You are making silly funny
>>> s.silly
You are getting silly
'funny'
>>> del s.silly
Whoah, you killed silly!
>>> help(SillyDecorated)
Help on class SillyDecorated in module __main__:

class SillyDecorated(builtins.object)
 |  Data descriptors defined here:
 |
 |  __dict__
 |      dictionary for instance variables (if defined)
 |
 |  __weakref__
 |      list of weak references to the object (if defined
)
 |
 |  silly
 |      This is a silly property
```

When to use properties

There are some factors to take into account when deciding to use a property.

- Technically, in Python, data, properties and methods are all attributes of a class.
- Methods are callable attributes, and properties are customisable attributes.
- Methods should typically represent actions performed by the object. When you call a method, it should do something. Method names are generally verbs.
- However, a normal attribute is not an action.

When to use properties

How to decide between standard data attributes and properties?

- In general, always use a standard attribute until you need to control access to that property in some way.
- The only difference between an attribute and a property is that we can invoke custom actions automatically when a property is retrieved, set or deleted.

When to use properties

An example, to store a custom behavior locally to avoid expensive calculations (a network request or database query).

- Use a custom getter on the property
- Perform the lookup or expensive calculation for the first time the value is retrieved
- Then locally cache the value as a private attribute on our project
- For all other requests, we return the local stored data.
- This is a simple webpage cache.

When to use properties

```
from urllib.request import urlopen

class WebPage:
    def __init__(self, url):
        self.url = url
        self._content = None

    @property
    def content(self):
        if not self._content:
            print("Retrieving New Page...")
            self._content = urlopen(self.url).read()
        return self._content
```

We can test this code to see that the page is only retrieved once.

```
>>> webpage = WebPage("http://www.google.com")
>>> now = time.time()
>>> content1=webpage.content
Retrieving New Page...
>>> time.time()-now
13.1511070728302
>>> now=time.time()
>>> content2=webpage.content
>>> time.time()-now
10.11282992362976
>>> content2==content1
True
>>> print(content1)
b'<!doctype html><html itemscope="" itemtype="http://sche
ma.org/WebPage" lang="en-AU"><head><meta content="text/ht
ml; charset=UTF-8" http-equiv="Content-Type"><meta conten
t="/images/branding/googleg/1x/googleg_standard_color_128
```

Manager objects

- Manager objects are higher-level objects which manage other objects – the objects tie everything together.
- Manager objects are more like office managers.
- They don’t do the actual visible work but handle the communications between other objects.
- The attributes on a management class tend to refer to other objects that do the visible work; the behaviors on such a class delegate to those other classes at the right time, and pass passages between them.

Manager objects

- We will have a look a program that does a find-and-replace action for text files stored in a zip file.
- We will have objects to represent the zip file and each individual text file.
- The class initialized with the .zip filename, and search and replace strings.
- A temporary directory is created to store the unzipped files in.
- We will use the pathlib (Python build-in library) to helps out with file and directory manipulation
- The manager object will be responsible for ensuring
  - Unzipping the compressed file
  - Performing the find-and-replace action
  - Zipping up the new files

Manager objects

```
class ZipReplace:
    def __init__(self, filename, search_string, replace_string):
        self.filename = filename
        self.search_string = search_string
        self.replace_string = replace_string
        self.temp_directory = Path(f"unzipped-{filename}")

    def zip_find_replace(self):
        self.unzip_files()
        self.find_replace()
        self.zip_files()
```

- '__init__' initialize the object with arguments.
- 'zip_find_replace' is the manager method which delegates responsibility to other objects in three steps, i.e., unzip files, find replace, and zip files.

Manager objects

Advantages of having steps in one method:

- Readability: The code for each step is in a self-contained unit that is easy to read and understand. The method name describes what the method does, and less additional documentation is required to understand what is going on.
- Extensibility: If a subclass wanted to use compressed RAR files instead of ZIP files, it could override the zip and unzip methods without having to duplicate the find_replace method.
- Partitioning: An external class could create an instance of this class and call the find_replace method directly on some folder without having to zip the content.

Manager objects

```
    def unzip_files(self):
        self.temp_directory.mkdir()
        with zipfile.ZipFile(self.filename) as zip:
            zip.extractall(self.temp_directory)

    def find_replace(self):
        for filename in self.temp_directory.iterdir():
            with filename.open() as file:
                contents = file.read()
            contents = contents.replace(self.search_string, self.replace_string)
            with filename.open("w") as file:
                file.write(contents)

    def zip_files(self):
        with zipfile.ZipFile(self.filename, "w") as file:
            for filename in self.temp_directory.iterdir():
                file.write(filename, filename.name)
        shutil.rmtree(self.temp_directory)


if __name__ == "__main__":
    ZipReplace(*sys.argv[1:4]).zip_find_replace()
```

```
$python zipsearch.py hello.zip hello hi
```

Removing duplicate code

Why is duplicate code a bad thing?

- Readability
- If you're writing a new piece of code that is similar to an earlier piece, you can copy and change the old code and put the new code in a new location. Alternatively, you can write fresh code with similar behaviour.
- However, you may forget all the details and wonder why you need similar pieces of code when you revisit your code after a year.
- You will spend extra time to understand why and how they are different. This wastes your time.
- Code should always be written to be readable first.

Removing duplicate code

Why is duplicate code a bad thing?

- Maintainability
- Keeping similar pieces of code up to date can be a nightmare.
- You have to remember to update all sections whenever you update one of them.
- You have to remember how multiple sections differ.
- If you miss any sections, you may end up with annoying bugs and question marks in your mind: ‘I had fixed that already, why is it still happening?’
- What if your code is maintained by someone else, and you are asked to maintain other people’s code?
- Astronomical amounts of time will be spent to understand, compare and test the similar code.
- Therefore, as a programmer, you should write the code with a non-repetitive manner.

Don’t Repeat Yourself (DRY)

- This is why almost all programmers (not only Python programmer) follow the Don’t Repeat Yourself (DRY) principle.
- DRY code is maintainable code.
- It is suggested never use the copy-and-paste feature when you write your code.
- To remove the code duplication, we can define a function that accepts parameters to account for whatever parts are different.

Don’t Repeat Yourself (DRY) – in practice

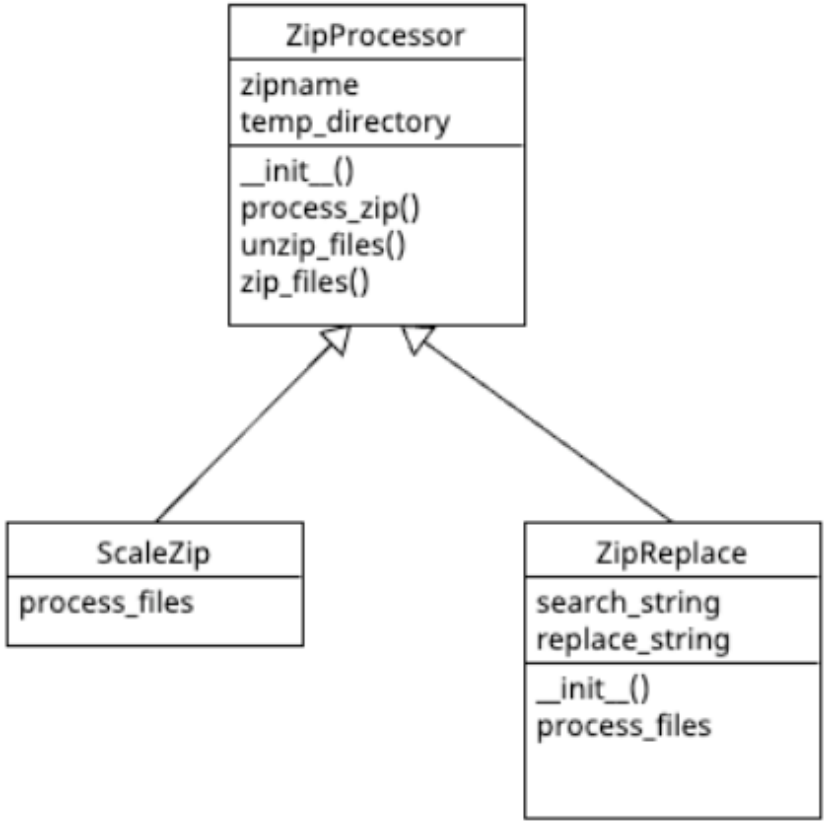
A better way to reuse existing code.

- Suppose we want to write another piece of code to scale all images in a zip fil to 640*480.
- Your first thought might be to reuse the ZipReplace class by copying and pasting, and change the find_replace method to scale_image method or something similar.
- However, this is not the optimal solution.
- What if we want to change the unzip and zip methods to open RAR files in the future? We’d have to change the code in both places.
- A better solution is to use the inheritance.

Don’t Repeat Yourself (DRY) – in practice

An inheritance-based solution.

- First, we modify our original ZipReplace class into a superclass (ZipProcessor) for processing generic zip files.
- We modify some method names in the superclass to avoid confusion and drop some parameters to the subclasses.
- ZipReplace class and ScaleZip class are the subclass which handle the actual jobs to process zip files for different purposes by using the polymorphism.



Don’t Repeat Yourself (DRY) – in practice

```
class ZipProcessor:
    def __init__(self, zipname):
        self.zipname = zipname
        self.temp_directory = Path(f"unzipped-{zipname[:-4]}")

    def process_zip(self):
        self.unzip_files()
        self.process_files()
        self.zip_files()

    def unzip_files(self):
        self.temp_directory.mkdir()
        with zipfile.ZipFile(self.zipname) as zip:
            zip.extractall(self.temp_directory)

    def zip_files(self):
        with zipfile.ZipFile(self.zipname, "w") as file:
            for filename in self.temp_directory.iterdir():
                file.write(filename, filename.name)
        shutil.rmtree(self.temp_directory)



if __name__ == "__main__":
    # ZipReplace(*sys.argv[1:4]).process_zip()
    ScaleZip(*sys.argv[1:4]).process_zip()
```

```
class ZipReplace(ZipProcessor):
    def __init__(self, filename, search_string, replace_string):
        super().__init__(filename)
        self.search_string = search_string
        self.replace_string = replace_string

    def process_files(self):
        """perform a search and replace on all files in the
        temporary directory"""
        for filename in self.temp_directory.iterdir():
            with filename.open() as file:
                contents = file.read()
            contents = contents.replace(self.search_string, self.replace_string)
            with filename.open("w") as file:
                file.write(contents)
```

```
class ScaleZip(ZipProcessor):
    def process_files(self):
        """Scale each image in the directory to 640x480"""
        for filename in self.temp_directory.iterdir():
            im = Image.open(str(filename))
            scaled = im.resize((640, 480))
            scaled.save(filename)
```

Casey study – a simple text editor

- In this case study, we will model a simple text editor as a Document class.
- When should I choose an object versus a built-in type?
- What objects, functions, or properties should the text editor have?

Casey study – a simple text editor

First question: how to represent the document contents.

- It is a text editor, so the contents are just a string.
- Can we just use strings type to represent the contents?
- The answer is yes, but has some issues.
- In Python (same as in Java), Strings are not mutable type. Once a string is defined, you can't change it.
- So you have to create a brand new string object with different content if you want to insert, change or delete any new letters.
- It works, but will leave a lot of string objects in the memory.
- Same as Java, Python also has a garbage collector to clean up the garbage objects in the member.
- So, instead of a string, we will use a list of characters to represent the text content. Then we can modify at will。

Casey study – a simple text editor

Possible things we might want to do with a text editor.

- Inserting
- Deleting
- Selecting characters
- Cutting
- Copying
- Pasting
- Saving
- Closing and opening

Casey study – a simple text editor

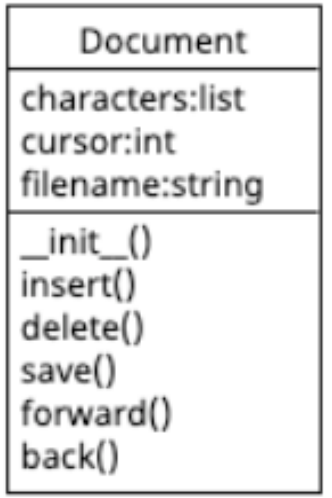
A pertinent question is:

- should this class be composed of a bunch of basic Python objects such as str filenames, int cursor positions, and a list of characters?
- Or should some or all of those things be specially defined objects in their own right?
- What about individual lines and characters? Do they need to have classes of their own?

We need answer these questions. Before that, we can start with the simplest possible Document class first with some essential functions.

Casey study – a simple text editor

```
class Document:
    def __init__(self):
        self.characters = []
        self.cursor = 0
        self.filename = ''

    def insert(self, character):
        self.characters.insert(self.cursor, character)
        self.cursor+=1

    def delete(self):
        del self.characters[self.cursor]

    def save(self):
        with open(self.filename, 'w') as f:
            f.write(''.join(self.characters))

    def forward(self):
        self.cursor+=1

    def back(self):
        self.cursor-=1
```



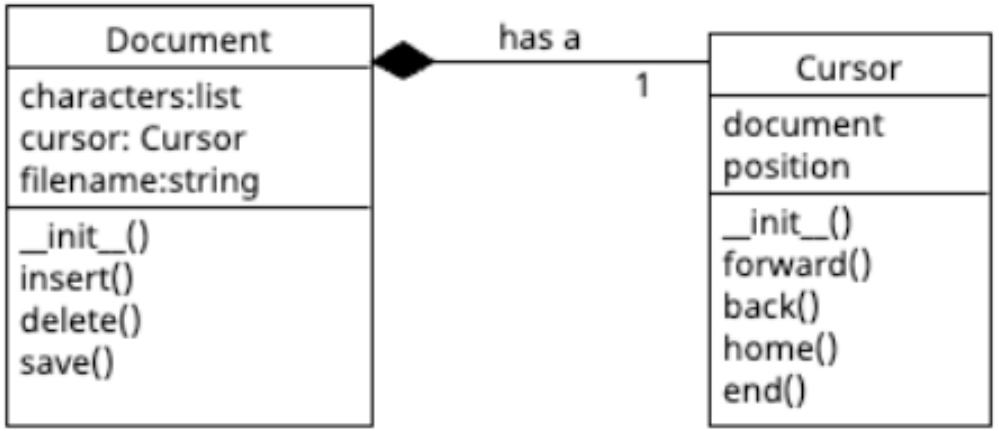

```
>>> doc = Document()
>>> doc.filename="test_document"
>>> doc.insert('h')
>>> doc.insert('i')
>>> "".join(doc.characters)
'hi'
>>> doc.back()
>>> doc.delete()
>>> doc.insert('a')
>>> "".join(doc.characters)
'ha'
```

Casey study – a simple text editor

- It looks like it's working. We could connect a keyboard's letter and arrow keys to these methods and the document would track everything just fine.
- But what if we want to connect more than just arrow keys.
- What if we want to connect the Home and End keys as well?
- We could add more methods to the Document class that search forward or backward for newline characters (a newline character, escaped as \n, represents the end of one line and the beginning of a new one) in the string and jump to them.
- But if we did that for every possible movement action (move by words, move by sentences, Page Up, Page Down, end of line, beginning of white space, and others), the class would be huge.
- Maybe it would be better to put those methods on a separate object.

Casey study – a simple text editor

- So, let's turn the Cursor attribute into an object that is aware of its position and can manipulate that position.
- The relationship between Document class and the Cursor class could be a composition, i.e., a document object contains a cursor object
- Forward and back methods can be move to the Cursor class because they will modify the position of the cursor object.
- Then we can also add more functions, such home and end methods.



Casey study – a simple text editor

```
class Cursor:
    def __init__(self, document):
        self.document = document
        self.position = 0

    def forward(self):
        self.position += 1

    def back(self):
        self.position -= 1

    def home(self):
        while self.document.characters[self.position-1] != "\n":
            self.position -= 1
            if self.position == 0:
                # Got to beginning of file before newline
                break

    def end(self):
        while (
            self.position < len(self.document.characters)
            and self.document.characters[self.position] != "\n"
        ):
            self.position += 1
```

Casey study – a simple text editor

```
class Document:
    def __init__(self):
        self.characters = []
        self.cursor = Cursor(self)
        self.filename = ""

    def insert(self, character):
        self.characters.insert(self.cursor.position, character)
        self.cursor.forward()

    def delete(self):
        del self.characters[self.cursor.position]

    def save(self):
        with open(self.filename, "w") as f:
            f.write("".join(self.characters))
```

```
>>> doc = Document()
>>> doc.insert('h')
>>> doc.insert('i')
>>> doc.insert('\n')
>>> doc.insert('w')
>>> doc.insert('o')
>>> doc.insert('r')
>>> doc.insert('l')
>>> doc.insert('d')
>>> doc.cursor.home()
>>> doc.insert('*')
>>> doc.cursor.end()
>>> doc.insert('!')
>>> print("".join(doc.characters))
hi
*world!
```

Casey study – a simple text editor

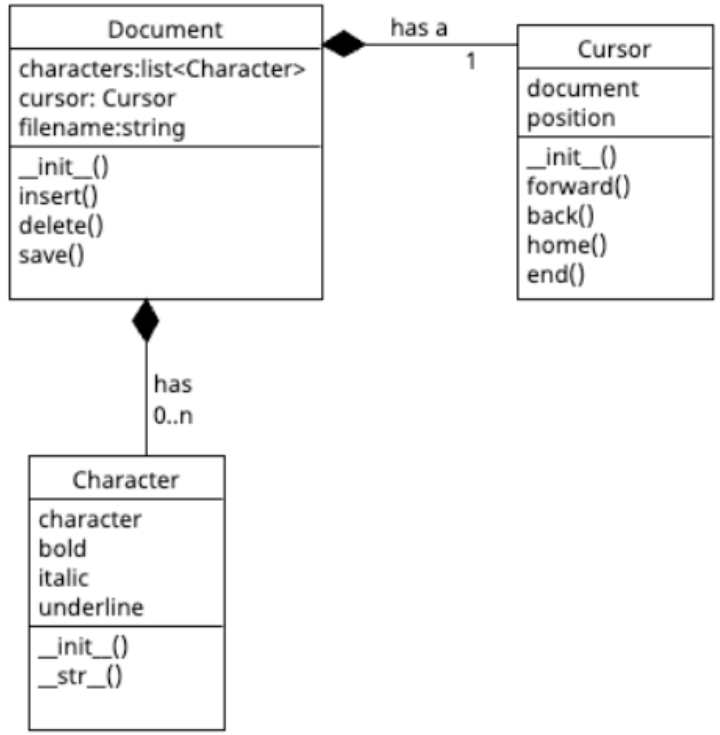
- As we've been using string join function a lot to concatenate the characters, we can add a property to the Document class.

```
@property
def string(self):
    return "".join((str(c) for c in self.characters))
```

```
>>> doc = Document()
>>> doc.insert('h')
>>> doc.insert('i')
>>> doc.insert('\n')
>>> print(doc.string)
    hi
```

Casey study – a simple text editor

- The very last extension is to enrich the text.
- Text can have bold, underlined or italic characters.
- We will add information to each character, indicating what formatting it should have.
- To do that, we need a class for characters. This class has an attribute representing the character as well as three Boolean attributes representing whether it is bold, italic or underlined.
- We will use prefixes to indicate different formats, i.e., * for bold, / for italic, and _ for underline.
- We will override the __str__() to convert the text together with the formatting prefixes to a string.



Casey study – a simple text editor

```
class Character:
    def __init__(self, character, bold=False, italic=False, underline=False):
        assert len(character) == 1
        self.character = character
        self.bold = bold
        self.italic = italic
        self.underline = underline

    def __str__(self):
        bold = "*" if self.bold else ""
        italic = "/" if self.italic else ""
        underline = "_" if self.underline else ""
        return bold + italic + underline + self.character
```

Casey study – a simple text editor

```
class Document:
    def __init__(self):
        self.characters = []
        self.cursor = Cursor(self)
        self.filename = ""

    def insert(self, character):
        if not hasattr(character, "character"):
            character = Character(character)
        self.characters.insert(self.cursor.position, character)
        self.cursor.forward()

    def delete(self):
        del self.characters[self.cursor.position]

    def save(self):
        with open(self.filename, "w") as f:
            f.write("".join(self.characters))

    @property
    def string(self):
        return "".join((str(c) for c in self.characters))
```

Casey study – a simple text editor

```
class Cursor:
    def __init__(self, document):
        self.document = document
        self.position = 0

    def forward(self):
        self.position += 1

    def back(self):
        self.position -= 1

    def home(self):
        while self.document.characters[self.position - 1].character != "\n":
            self.position -= 1
            if self.position == 0:
                # Got to beginning of file before newline
                break

    def end(self):
        while (
            self.position < len(self.document.characters)
            and self.document.characters[self.position].character != "\n"
        ):
            self.position += 1
```

Casey study – a simple text editor

```
>>> doc = Document()
>>> doc.insert('h')
>>> doc.insert(Character('i',bold=True))
>>> doc.insert('\n')
>>> doc.insert(Character('w',italic=True))
>>> doc.insert(Character('o',italic=True))
>>> doc.insert(Character('r',underline=True))
>>> doc.insert('l')
>>> doc.insert('d')
>>> print(doc.string)
h*i
/w/o_rld
```

Suggested reading

Python 3 Object-Oriented Programming

- Chapter 5: When to Use Object-Oriented Programming

Python

- https://www.python.org/