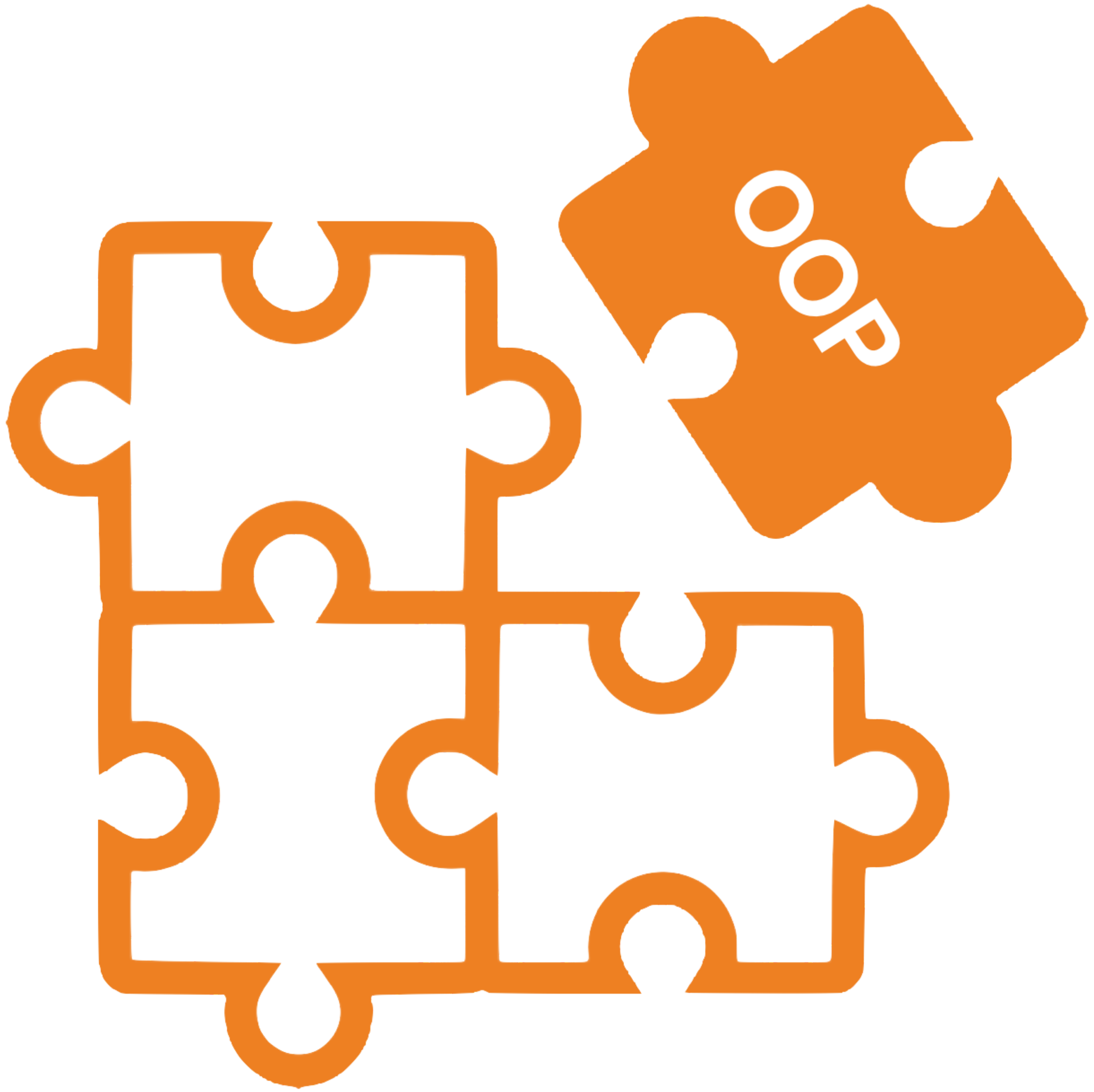
Adding Classes to a Package

SOFTWARE ENGINEERING FOR DATA SCIENTISTS IN PYTHON

Adam Spannbauer
Machine Learning Engineer at Eastman

Object oriented programming



2. Object oriented programming
We'll now look at how you can use classes to strengthen your package's functionality. Object Oriented Programming, or OOP, is a great way to write modular code, and reap the benefits of modularity such as easily understood and extensible code. We'll cover some aspects of OOP, but if you want a deeper dive I recommend the dedicated DataCamp courses. Let's jump into some code.

Anatomy of a class

working in `work_dir/my_package/my_class.py`

```
# Define a minimal class with an attribute
class MyClass:
    """A minimal example class

    :param value: value to set as the ``attribute`` attribute
    :ivar attribute: contains the contents of ``value`` passed in init
    """

    # Method to create a new instance of MyClass
    def __init__(self, value):
        # Define attribute with the contents of the value param
        self.attribute = value
```

3. Anatomy of a class
In python, OOP can be utilized by writing classes. Here we see a minimal class implementation. To start we use the keyword class followed by the name of our class. According to PEP8, our class name should be written in camel case, that is, our name should start with a capital letter and every word in the name should have a capital letter as well. **Unlike function and package names, class names should never contain underscores.** Next, is some documentation that will appear when a user calls help on our class... Last, we see a function with a familiar name, __init__. Similarly to your package's __init__.py file, this will initialize everything when a user wants to leverage your class.

Using a class in a package

working in `work_dir/my_package/__init__.py`

```
from .my_class import MyClass
```

working in `work_dir/my_script.py`

```
import my_package

# Create instance of MyClass
my_instance = my_package.MyClass(value='class attribute value')

# Print out class attribute value
print(my_instance.attribute)
```

```
'class attribute value'
```

4. Using a class in a package
To make our class easily accessible we can add it to our init file just like we've done with functions before. We use relative import syntax to import MyClass from the '**my class' dot py** file in the same directory, we can now import the package and access MyClass easily. Now let's create what's known as an instance of MyClass. To do this we call MyClass like a function and supply a string to the value parameter. Calling our class like this tells Python that we want to create an instance of our class by using the init method. Recall that MyClass's init method set the contents of value to a variable we referenced as self dot attribute. Users can access this attribute by referencing my underscore instance dot attribute. Note, nowhere in this script do we see the self-object that we used when defining our class.

The self convention

working in `work_dir/my_package/my_class.py`

```
# Define a minimal class with an attribute
class MyClass:
    """A minimal example class

    :param value: value to set as the ``attribute`` attribute
    :ivar attribute: contains the contents of ``value`` passed in init
    """

    # Method to create a new instance of MyClass
    def __init__(self, value):
        # Define attribute with the contents of the value param
        self.attribute = value
```

5. The self convention
Let's look more in depth at the use of self in our class's init method. Self is, in essence, a way to refer to a class instance even though we don't know what the user is actually going to name their instance. When defining typical class instance methods, like __init__, self is the first argument. However, if you recall creating an instance of MyClass, when using __init__, the user doesn't need to pass a value to this self argument; this is done automatically behind the scenes. Once in the method body, we can use self to access or define attributes. The user can then access these attributes by using their class's name in place of self, like we did in our script. Technically we can use a different word than self, but this a very strong PEP8 convention and not abiding by it will make your code very hard to read by your collaborators.

```
my_instance = my_package.MyClass(value='class attribute value')
```

Let's Practice

SOFTWARE ENGINEERING FOR DATA SCIENTISTS IN PYTHON

Leveraging Classes

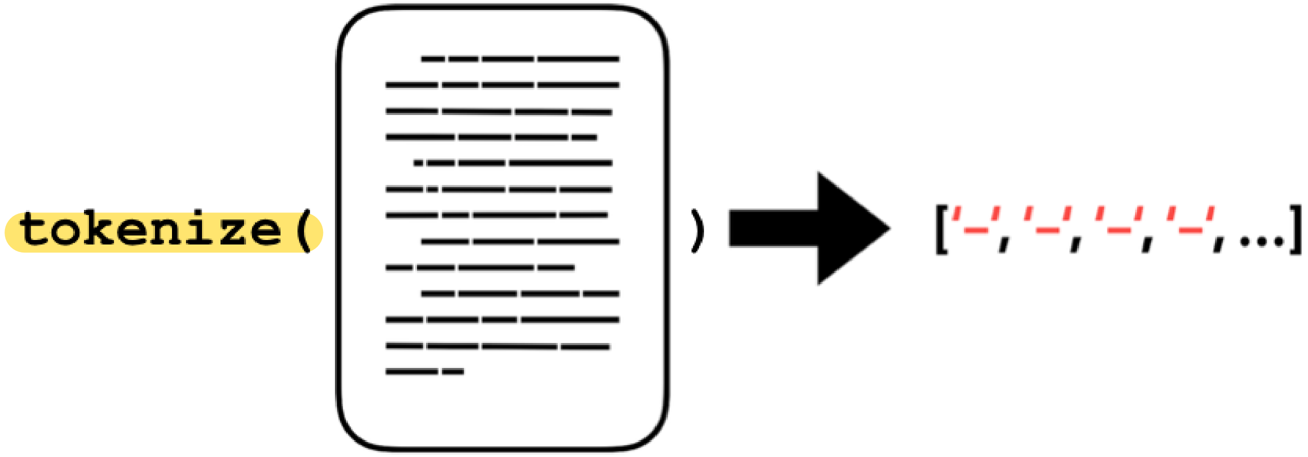
SOFTWARE ENGINEERING FOR DATA SCIENTISTS IN PYTHON

Adam Spannbauer
Machine Learning Engineer at Eastman

datacamp

Extending Document class

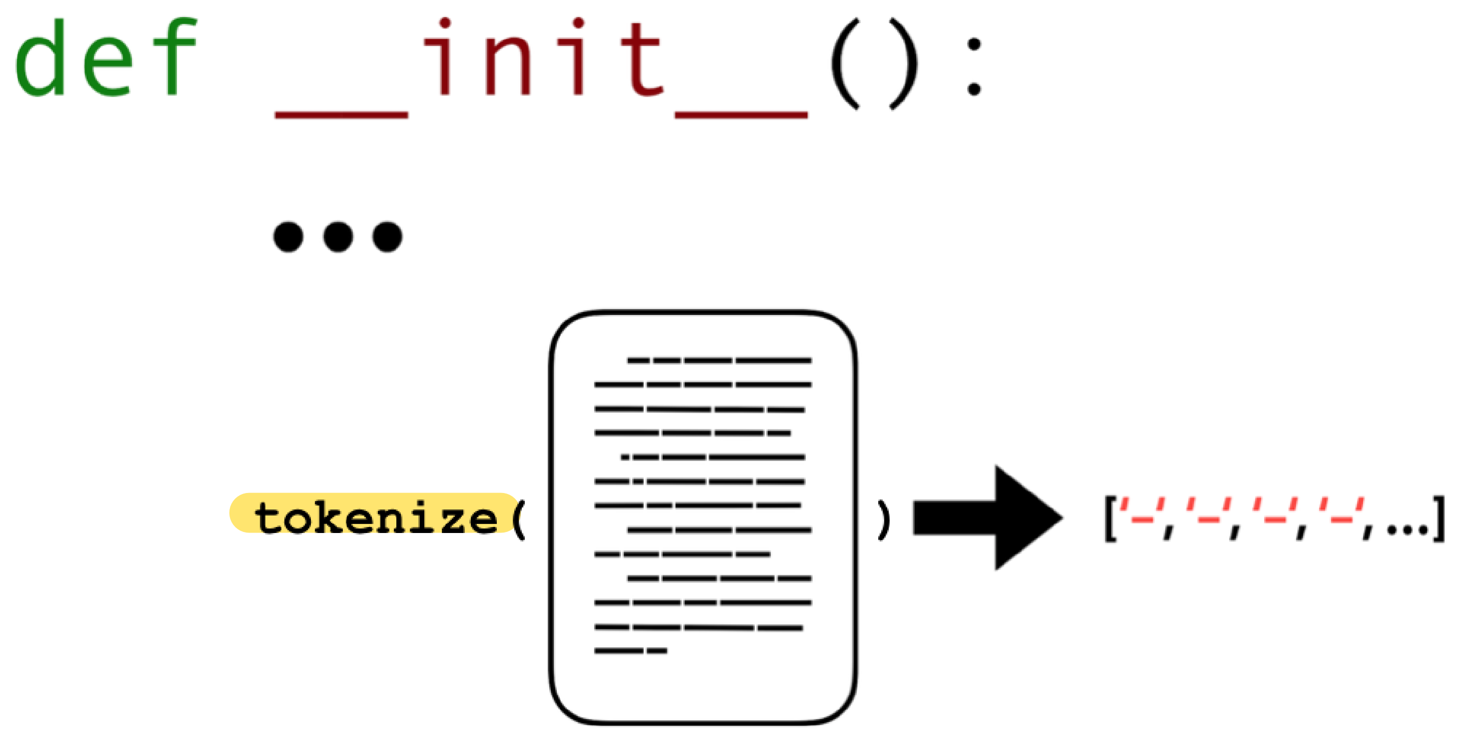
```python
class Document:
    def __init__(self, text):
        self.text = text
```



2. Extending Document class
Let's look at our current definition of Document and talk about how we can improve its functionality. Right now the class is just a container for the user provided text; this doesn't add much value for our user. In order for our Document class to be more useful, we can add more attributes & methods besides __init__. For example, let's say that in our workflow we always want to tokenize our documents as a first step. Tokenization is a common step in text analysis, it is the process of breaking up a document into individual words, also known as tokens.

Current document class

```python
class Document:
    def __init__(self, text):
        self.text = text
```



3. Current document class
As we've learned, __init__ is what's called when a user wants to create an instance of Document. This would be a convenient location to put a tokenization step. Placing the tokenization process inside of __init__ will ensure our Document is tokenized as soon as it is created, and this will save your user's the trouble of thinking about this step.

Revising __init__

```python
class Document:
    def __init__(self, text):
        self.text = text
        self.tokens = self._tokenize()

doc = Document('test doc')
print(doc.tokens)
```

4. Revising __init__
Our new __init__ method might look something like this. You can see that we added a line that calls self dot underscore tokenize and dumps the output into an attribute named tokens.
So where does underscore tokenize come from? and why does it have an underscore in front of it?

```
['test', 'doc']
```

Adding _tokenize() method

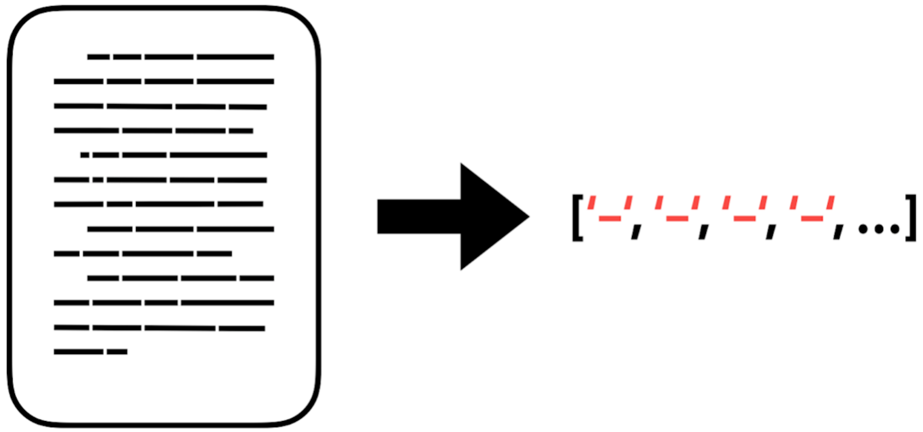
```
# Import function to perform tokenization
from .token_utils import tokenize
class Document:
    def __init__(self, text, token_regex=r'[a-zA-z]+'):
        self.text = text
        self.tokens = self._tokenize()
    def _tokenize(self):
        return tokenize(self.text)
```

5. Adding _tokenize() method

Since this course isn't focused on teaching text analytics, we aren't going to cover the tokenization function implementation; we'll simply import a function to do it for us. ... Moving on to the definition of underscore tokenize. We only pass one parameter to the function, the prescribed self convention that will represent an instance of the Document object. Since the tokenize function is already written, all that's left to do is call it on the text attribute. And voila! The tokenization process will now be completed automatically as soon as a user creates a Document instance. So why did we use a leading underscore when naming _tokenize?

Non-public methods

```
def __init__():
    ...
```



6. Non-public methods

The reason we added the tokenization process to the __init__ method is that we wanted tokenization to happen immediately without the user having to think about it. Because of this, the user doesn't need to call underscore tokenize themselves; in other words, this method doesn't need to be 'public' to the user. According to PEP 8, non-public methods should be named with a single leading underscore. This signifies to the user that the method is intended for internal use only. Users can still use non-public methods in their own workflow, but they must do so at their own risk since the developer did not intend for them to do so.

The risks of non-public methods

- Lack of documentation
- Unpredictability

7. The risks of non-public methods

The risks of using a non-public method in your own workflow include: little or no documentation and the function's input or output might change without warning when the developer updates their package.

Let's Practice

SOFTWARE ENGINEERING FOR DATA SCIENTISTS IN PYTHON

Classes and the DRY principle

SOFTWARE ENGINEERING FOR DATA SCIENTISTS IN PYTHON

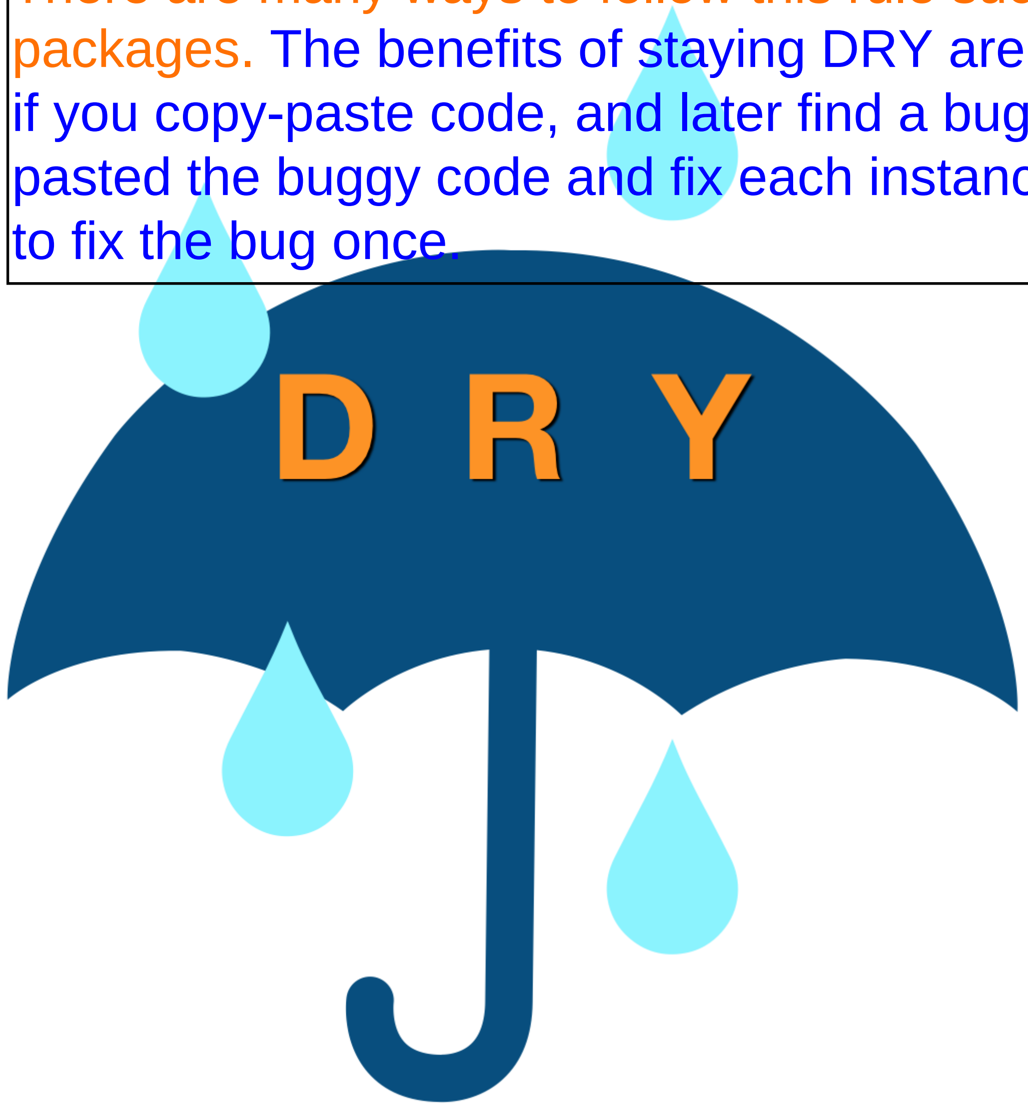
Adam Spannbauer
Machine Learning Engineer at Eastman

Creating a SocialMedia Class

2. Creating a SocialMedia Class
To do this let's make a SocialMedia class that performs the same functions as the Document class, but can additionally analyze social media specific things like hashtags. However, when making this SocialMedia class we want to preserve the Document class as is to be used for more general analysis. So, how can we do this? A first guess might be to copy-paste the contents of the Document class. This violates what is known as the DRY principle in Software Engineering.

The DRY principle

3. The DRY principle
The DRY principle is a great rule that, if followed, can help you write modular, readable code. DRY stands for: Don't Repeat Yourself.
There are many ways to follow this rule such as writing re-usable functions, classes, and packages. The benefits of staying DRY are not only saving time by reusing code, but also if you copy-paste code, and later find a bug, you have to remember everywhere you've pasted the buggy code and fix each instance individually. If you stay DRY you only need to fix the bug once.



The DRY principle

The DRY principle

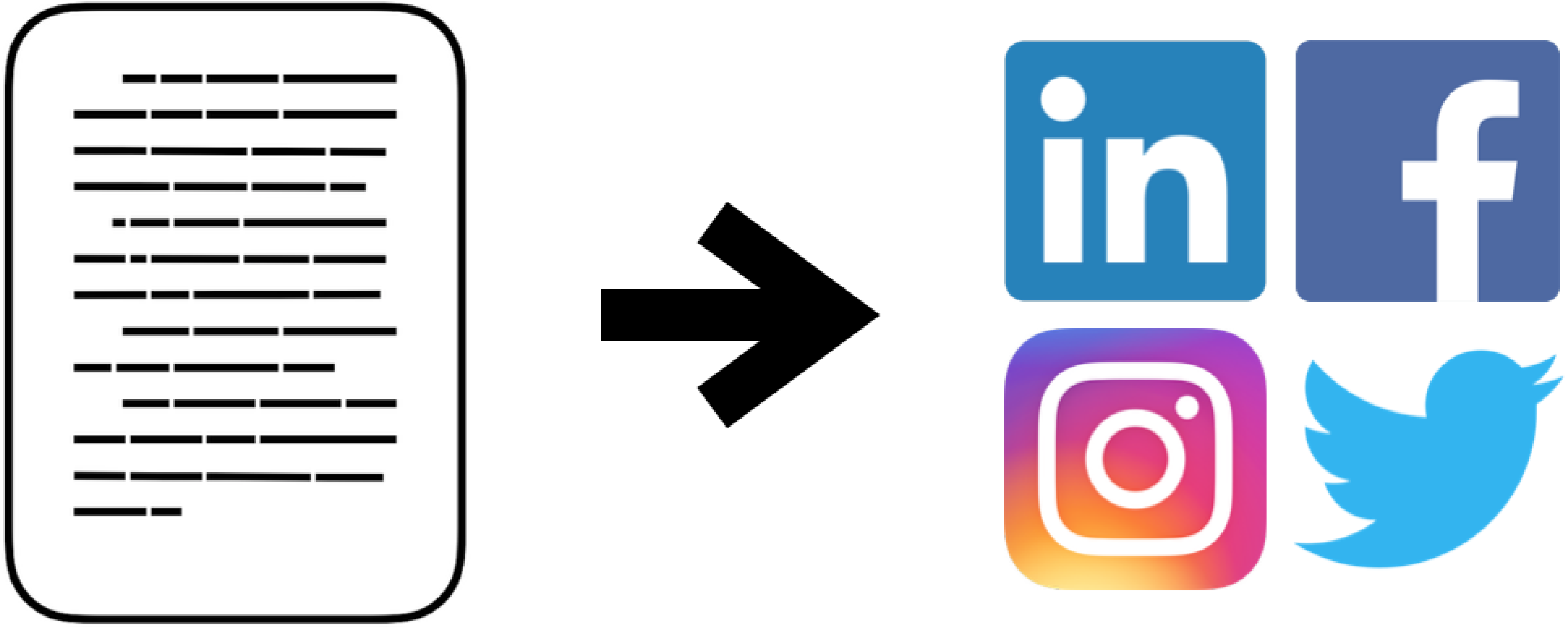
The DRY principle

Intro to inheritance

7. Intro to inheritance

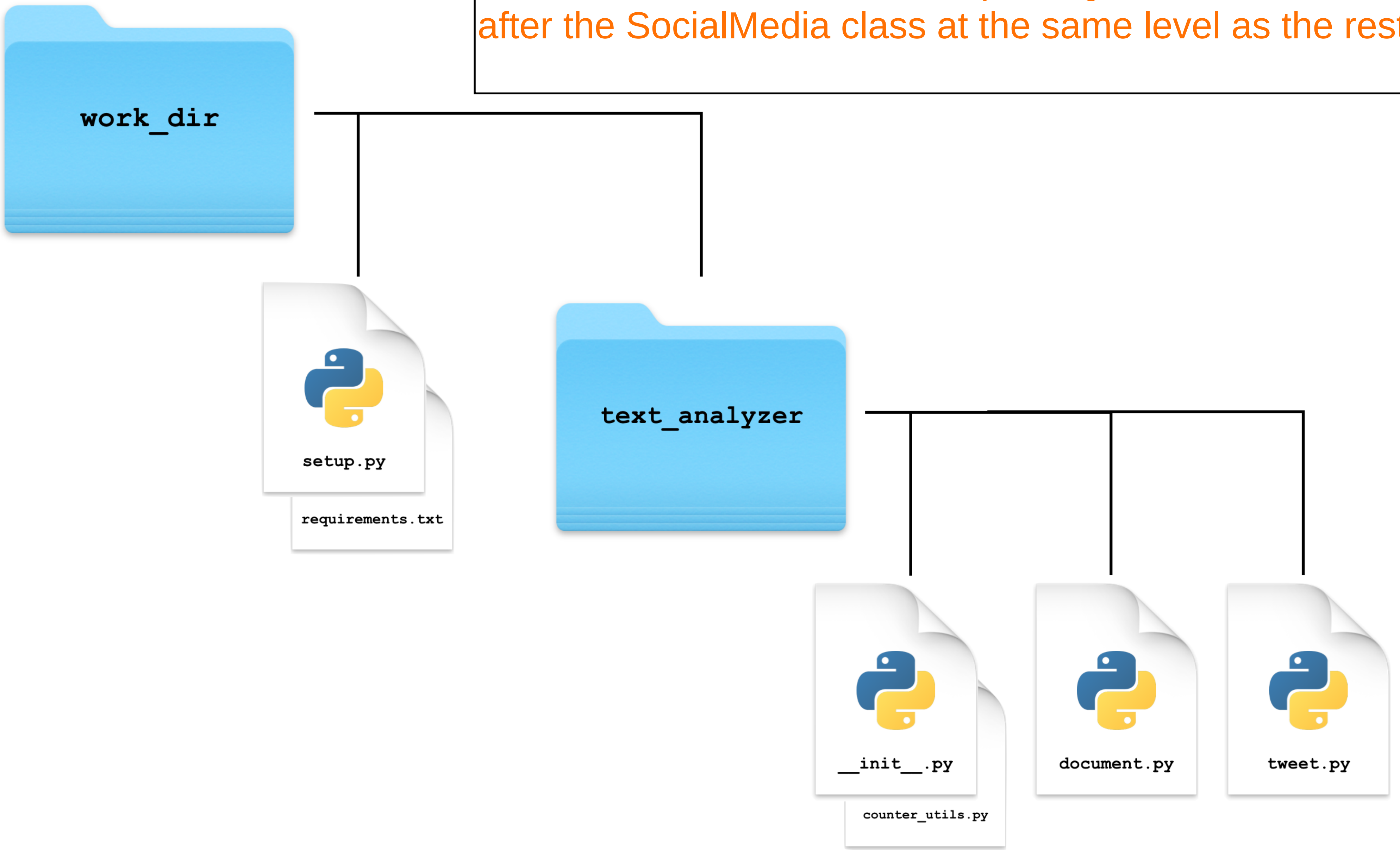
In the case of extending the Document class to be a SocialMedia class, we can stay DRY by using the Object Oriented Programming concept of inheritance. With inheritance, we start with a parent class and we pass on it's functionality to a child class. The child class inherits all the methods and attributes of its parent, and we're able to add additional functionality without affecting the parent class.

Parent Class	Child Class
`Document`	`SocialMedia`



Inheritance in Python

8. Inheritance in Python
So how do we leverage inheritance in Python? To start let's see how the child SocialMedia class fits into the package structure. It will live as a single file named after the SocialMedia class at the same level as the rest of the package's code.



Inheritance in Python

```python
# Import ParentClass object
from .parent_class import ParentClass

# Create a child class with inheritance
class ChildClass(ParentClass):
    def __init__(self):
        # Call parent's __init__ method
        ParentClass.__init__(self)
        # Add attribute unique to child class
        self.child_attribute = "I'm a child class attribute!"

# Create a ChildClass instance
child_class = ChildClass()
print(child_class.child_attribute)
print(child_class.parent_attribute)
```

9. Inheritance in Python

Now let's see how to actually write a child class that uses inheritance. First, we import the ParentClass for use in defining our ChildClass. To let Python know we're using inheritance, we pass the ParentClass as an argument in our class statement. Last, we call our ParentClass's __init__ method. Remember, __init__ builds an instance of a class and it also accepts self as its first argument. With this call, we're building an instance of ParentClass and storing it right back into self. This means that self now has all the methods and attributes that an instance of ParentClass would. We can now use self as normal to build in additional functionality unique to ChildClass. Here, we just add an attribute. With our definitions, we can now create an instance of our new ChildClass as we've seen before, and then access attributes from both the parent and the child.

Using inheritance like this can lead to short, easy to read definitions of children classes that are jam-packed with the functionality of their parents.

Let's Practice

SOFTWARE ENGINEERING FOR DATA SCIENTISTS IN PYTHON

Multilevel inheritance

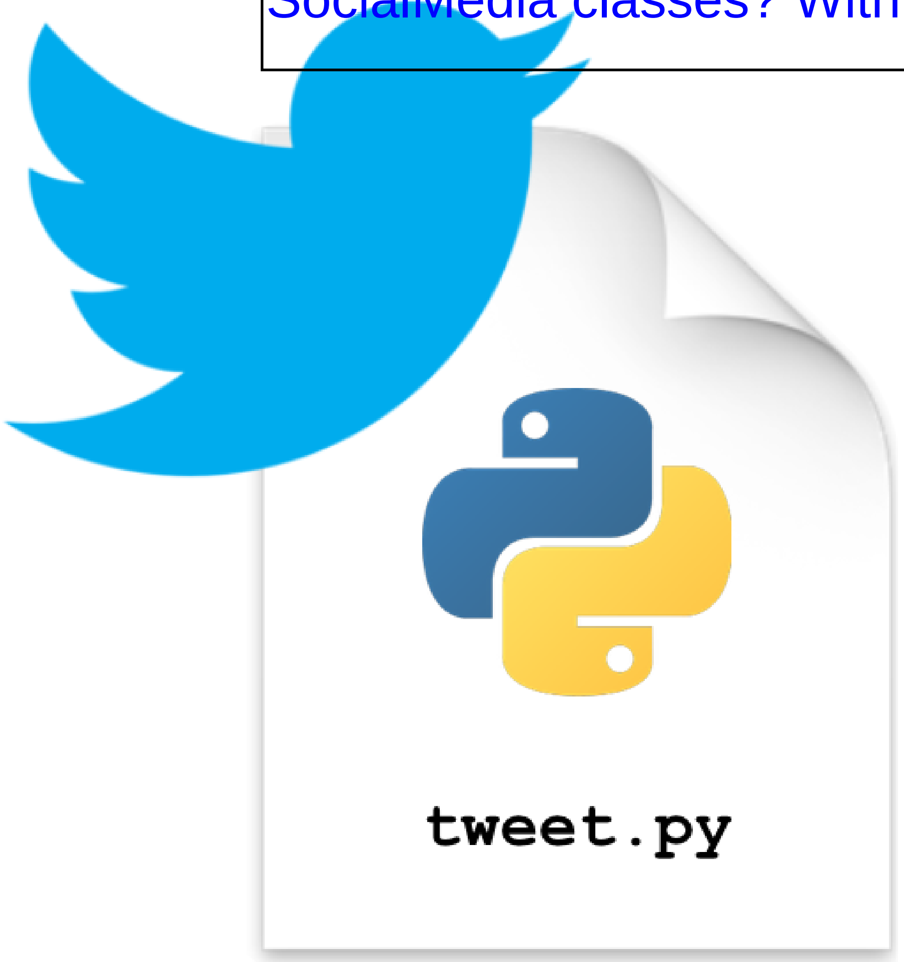
SOFTWARE ENGINEERING FOR DATA SCIENTISTS IN PYTHON

Adam Spannbauer
Machine Learning Engineer at Eastman

Creating a Tweet class

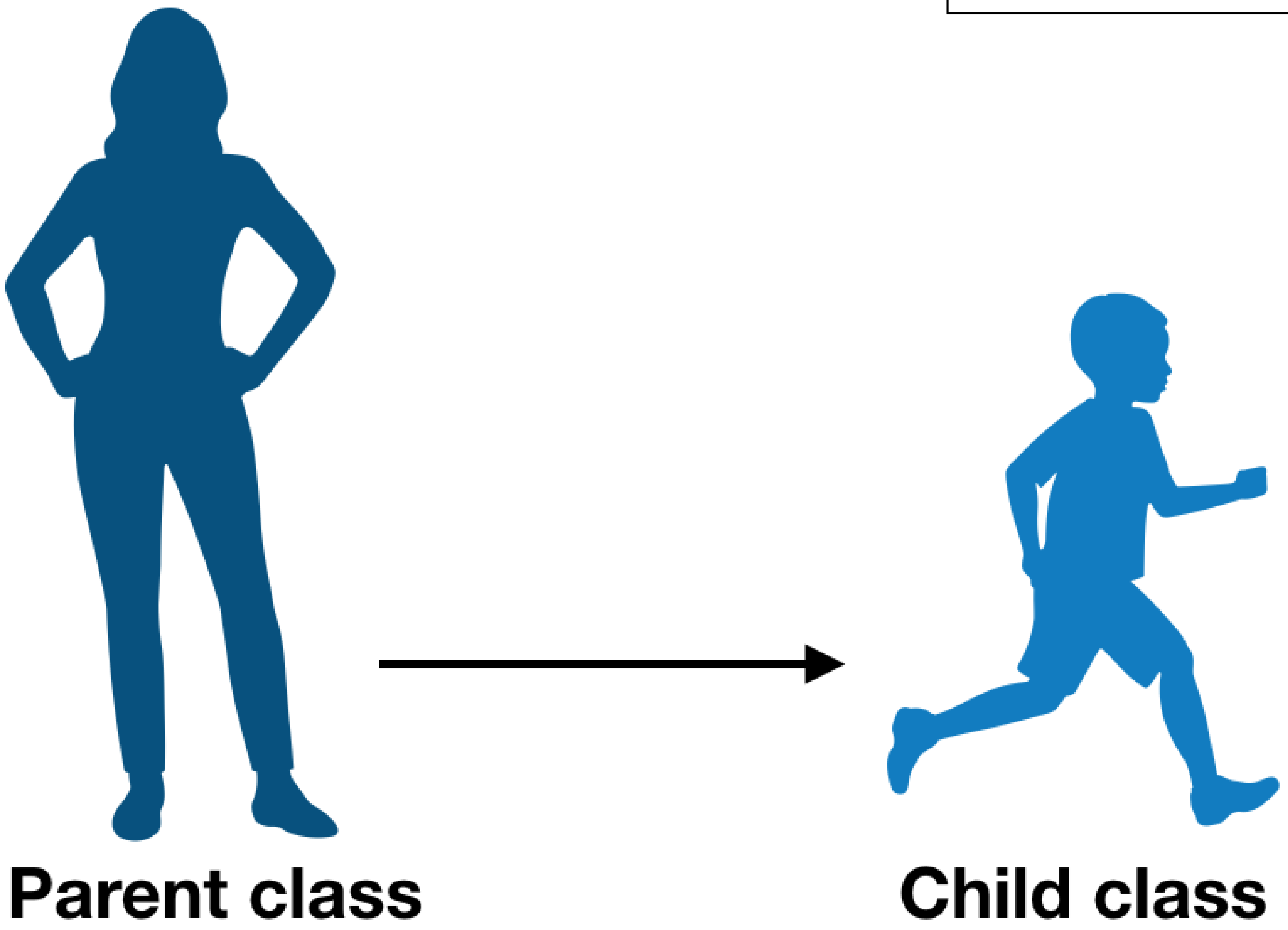
2. Creating a Tweet class

The SocialMedia class has some general functionality dealing with hashtags and mentions that works across multiple social media platforms. If we wanted to include functionality about retweets, it wouldn't be appropriate to include in the general class; instead, we can create a Tweet class. So how can we do this without losing the benefits of both the Document and SocialMedia classes? With inheritance again of course!



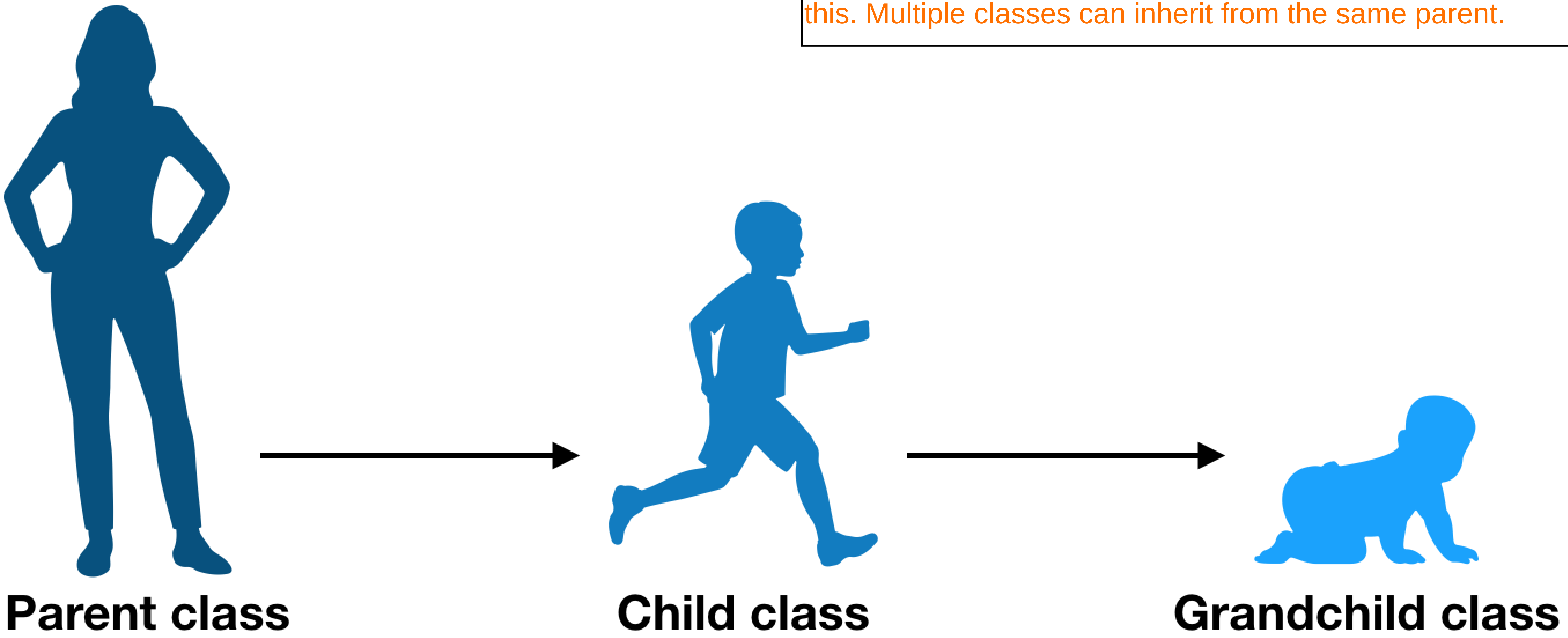
Multilevel inheritance

3. Multilevel inheritance
We've seen before that a child class can inherit from a parent class. We can continue the family tree again with inheritance. With multilevel inheritance, our child class is all grown up and can its functionality onto its children.



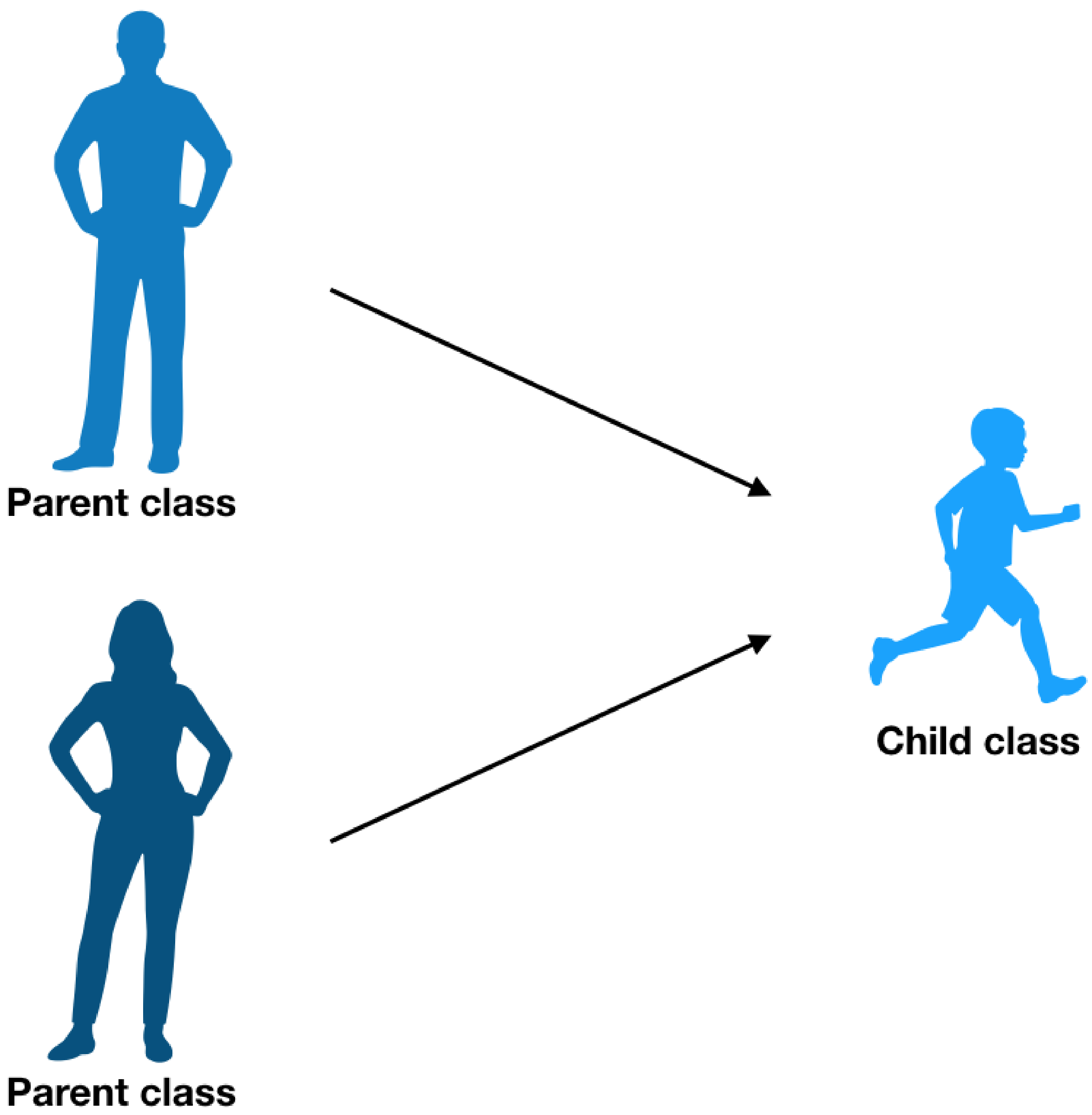
Multilevel inheritance

4. Multilevel inheritance
Thanks to inheritance we pass along the traits of all prior classes in the family tree. Note, that in the graphic we only have one inheriting class at each level, but we are by no means limited to this. Multiple classes can inherit from the same parent.



Multiple inheritance

5. Multiple inheritance
In fact, one child class can inherit from multiple parents. This more advanced OOP concept is known as 'multiple



Multilevel inheritance and super

```python
class Parent:
    def __init__(self):
        print("I'm a parent!")


class Child(Parent):
    def __init__(self):
        Parent.__init__()
        print("I'm a child!")

class SuperChild(Parent):
    def __init__(self):
        super().__init__()
        print("I'm a super child!")
```

6. Multilevel inheritance and super
Let's code up an example of multilevel inheritance. We'll start with the inheritance pattern we've seen before. Here we define a Parent and Child class that inherits from the Parent. We could do this differently, by using the super function. Instead, of directly calling the __init__ method of Parent we use the super function. This makes no functional difference in our code here but it has some advantages in maintainability and when implementing multiple inheritance. However, there are some 'gotchas' that can arise with super and multiple inheritance; you can check out this companion DataCamp tutorial to learn more about it.

Learn more about multiple inheritance & `super()` .

Multilevel inheritance and super

```python
class Parent:
    def __init__(self):
        print("I'm a parent!")

class SuperChild(Parent):
    def __init__(self):
        super().__init__()
        print("I'm a super child!")

class Grandchild(SuperChild):
    def __init__(self):
        super().__init__()
        print("I'm a grandchild!")

grandchild = Grandchild()
```

7. Multilevel inheritance and super
Let's continue the family tree, to create a Grandchild class that inherits from SuperChild we use the same super syntax in its __init__ method, and voila. If we create an instance of Grandchild we can see from the output that each __init__ method in the family tree was called.

```
I'm a parent!
I'm a super child!
I'm a grandchild!
```

Keeping track of inherited attributes

```
# Create an instance of SocialMedia
sm = SocialMedia('@DataCamp #DataScience #Python #sklearn')
# What methods does sm have?  ¯\_(?)_/¯
dir(sm)
```

```
['__class__', '__delattr__', '__dict__', '__dir__', '__doc__', '__eq__',
'__format__', '__ge__', '__getattribute__', '__gt__', '__hash__', '__init__',
'__init_subclass__', '__le__', '__lt__', '__module__', '__ne__', '__new__',
'__reduce__', '__reduce_ex__', '__repr__', '__setattr__', '__sizeof__',
'__str__', '__subclasshook__', '__weakref__', '_count_hashtags',
'_count_mentions', '_count_words', '_tokenize', 'hashtag_counts',
'mention_counts', 'text', 'tokens', 'word_counts']
```

8. Keeping track of inherited attributes

If you're using inheritance it's sometimes hard to remember what attributes and methods your class has at the end of it all. If you're using an IDE, then generally you can use tab complete to get a list of suggestions. However, if you want to get the info from the console, you can use either help or the dir function. help is a good option in most cases, but it will only include public methods in it's output. Using dir will print a fairly exhaustive list of what your class has under the covers. The list includes methods that come with our class object by default. Near the end of the list, you see all the methods and attributes that you personally programmed into the SocialMedia class. dir can definitely come in handy, but a warning from the documentation, "dir is supplied primarily as a convenience for use at an interactive prompt." So it's not advised to use it in your scripts.

Let's Practice

SOFTWARE ENGINEERING FOR DATA SCIENTISTS IN PYTHON