



Basics for NLP

Classes & Pair Programming

Iulia, Rishu, & Chris

BRACE YOURSELVES

HOW CAN I CREATE A FUNCTION THAT
CALLS ANOTHER FUNCTION AND PASSES
IN AN INSTANCE ATTRIBUTE AS AN ARGUMENT?

Summary

- Classes
 - What are classes?
 - Why use classes?
 - What are the parts of a class?
- Advanced Classes
 - What is inheritance?
 - What are Dunders?
 - What are Decorators?

Who needs class?!

What is a class?

- A class is an object that has:
 - Specific attributes
 - Specific behaviours
- Lists are a type of class in Python

Why use a class?

- They are basically a template in which you can modify, but:
 - They can share attributes between methods within them
 - They allow us to reuse code far more easily



Classes

What are classes?

Why use classes?

What are the parts of a class?

Examples


What makes a class?


- Parts of a class:
 - Instance creation
 - Class variables
 - Instance variables
 - Dunders?!



```
class Book():  
    page_width = 14  
    cover_color = "blue"  
  
    def __init__(self, name, writer, word_length):  
        self.name = name  
        self.writer = writer  
        self.word_length = word_length
```

What makes a class?

- Parts of a class:
 - Instance creation
 - Class variables 
 - Instance variables
 - Dunders?!



```
class Book():  
    page_width = 14  
    cover_color = "blue"  
  
    def __init__(self, name, writer, word_length):  
        self.name = name  
        self.writer = writer  
        self.word_length = word_length
```


What makes a class?


- Parts of a class:
 - Instance creation
 - Class variables
 - Instance variables
 - Dunders?!



```
class Book():  
    page_width = 14  
    cover_color = "blue"  
  
    def __init__(self, name, writer, word_length):  
        self.name = name  
        self.writer = writer  
        self.word_length = word_length
```


What makes a class?

- Parts of a class:
 - Instance creation
 - Class variables
 - Instance variables
 - Dunders?! 

```
class Book():  
    page_width = 14  
    cover_color = "blue"  
     __init__(self, name, writer, word_length):  
        self.name = name  
        self.writer = writer  
        self.word_length = word_length
```


Classes have two types of variables?!

Class variables

- These are declared within the class but outside of the function
- More general, apply to all instances of the class

Instance Variables

- These are declared inside the constructor which is the Dunder `__init__`
- More specific, prone to being unique



```
class Book():  
    page_width = 14  
    cover_color = "blue"  
  
    def __init__(self, name, writer, word_length):  
        self.name = name  
        self.writer = writer  
        self.word_length = word_length
```

Create an instance of a Book class

- If we do this, we should include the three instance based variables when creating it
 - Self.name
 - Self.writer
 - Self.wordlength
- What would the class variables be in this case?
 - Would they remain the same?
 - Would they change?

Create an instance of a Book class

- If we do this, we should include the three instance based variables when creating it
 - Self.name
 - Self.writer
 - Self.wordlength

```
b2 = Book("Machine Learning", "Jane Doe", 120000)
```

We have not specified the class variables while creating this instance. However, the b2 possesses these variables and we can access them.

```
b2.page_width  
14  
  
b2.cover_color  
'blue'
```

Create an instance of a Book class

- What if we wanted to change these class variables to something else?
 - Would we change them at the class level?
 - Can we change them elsewhere?

```
b2.cover_color = 'red'  
b2.cover_color  
'red'
```

Create an instance of a Book class

- What if we wanted to change these class variables to something else?
 - Would we change them at the class level?
 - Can we change them elsewhere?

```
Book.cover_color  
'blue'
```

Example

- What is the class called?
 - Class vars?
 - Instance vars?
 - What is the dunder?
- What do we put into the class to create one?
 - E.g. what variables
- What is Book.total_books doing?

```
class Book:
    total_books = 0
    book_types = ['Fiction', 'Non-Fiction']

    def __init__(self, title, author, book_type):
        self.title = title
        self.author = author
        self.book_type = book_type

        # Update class variables
        Book.total_books += 1

    def display_info(self):
        print(f>Title: {self.title}")
        print(f">Author: {self.author}")
        print(f">Book Type: {self.book_type}")
        print(f">Total Books: {Book.total_books}\n")
```

Example

- What would be the output of `total_books` for `book1` and `book2`?
 - Would this be the same after we set the class variable to 4?
 - What would `book3` and `book4` total books be?

```
book1 = Book("The Great Gatsby", "F. Scott Fitzgerald", "Fiction")
book2 = Book("Sapiens", "Yuval Noah Harari", "Non-Fiction")

# Change class variable total_books for book2
Book.total_books = 4

book1.display_info()
book2.display_info()

book3 = Book("Sapiens", "Yuval Noah Harari", "Non-Fiction")
book3.display_info()

book4 = Book("Sapiens", "Yuval Noah Harari", "Non-Fiction")
book4.display_info()
```


Example

- What would be the output of `total_books` for `book1` and `book2`?
 - Would this be the same after we set the class variable to 4?
 - What would `book3` and `book4` total books be?

```
book1 = Book("The Great Gatsby", "F. Scott Fitzgerald", "Fiction")
book2 = Book("Sapiens", "Yuval Noah Harari", "Non-Fiction")

# Change class variable total_books for book2
Book.total_books = 4

book1.display_info()
book2.display_info()

book3 = Book("Sapiens", "Yuval Noah Harari", "Non-Fiction")
book3.display_info()

book4 = Book("Sapiens", "Yuval Noah Harari", "Non-Fiction")
book4.display_info()
```

```
Title: The Great Gatsby
Author: F. Scott Fitzgerald
Book Type: Fiction
Total Books: 4
```

```
Title: Sapiens
Author: Yuval Noah Harari
Book Type: Non-Fiction
Total Books: 4
```

```
Title: Sapiens
Author: Yuval Noah Harari
Book Type: Non-Fiction
Total Books: 5
```

```
Title: Sapiens
Author: Yuval Noah Harari
Book Type: Non-Fiction
Total Books: 6
```

Example

- What would be the output of total_books be now for each?
 - Book1?
 - Book2?
 - Book3?
 - Book4?

```
book1 = Book("The Great Gatsby", "F. Scott Fitzgerald", "Fiction")
book2 = Book("Sapiens", "Yuval Noah Harari", "Non-Fiction")

# Change class variable total_books for book2
Book.total_books = 23

book1.display_info()

Book.total_books = 28
book2.display_info()

book3 = Book("Sapiens", "Yuval Noah Harari", "Non-Fiction")
book3.display_info()

Book.total_books = 2|

book4 = Book("Sapiens", "Yuval Noah Harari", "Non-Fiction")
book4.display_info()
```

Example

- What would be the output of total_books be now for each?
 - Book1?
 - Book2?
 - Book3?
 - Book4?

```
book1 = Book("The Great Gatsby", "F. Scott Fitzgerald", "Fiction")
book2 = Book("Sapiens", "Yuval Noah Harari", "Non-Fiction")

# Change class variable total_books for book2
Book.total_books = 23

book1.display_info()

Book.total_books = 28
book2.display_info()

book3 = Book("Sapiens", "Yuval Noah Harari", "Non-Fiction")
book3.display_info()

Book.total_books = 2

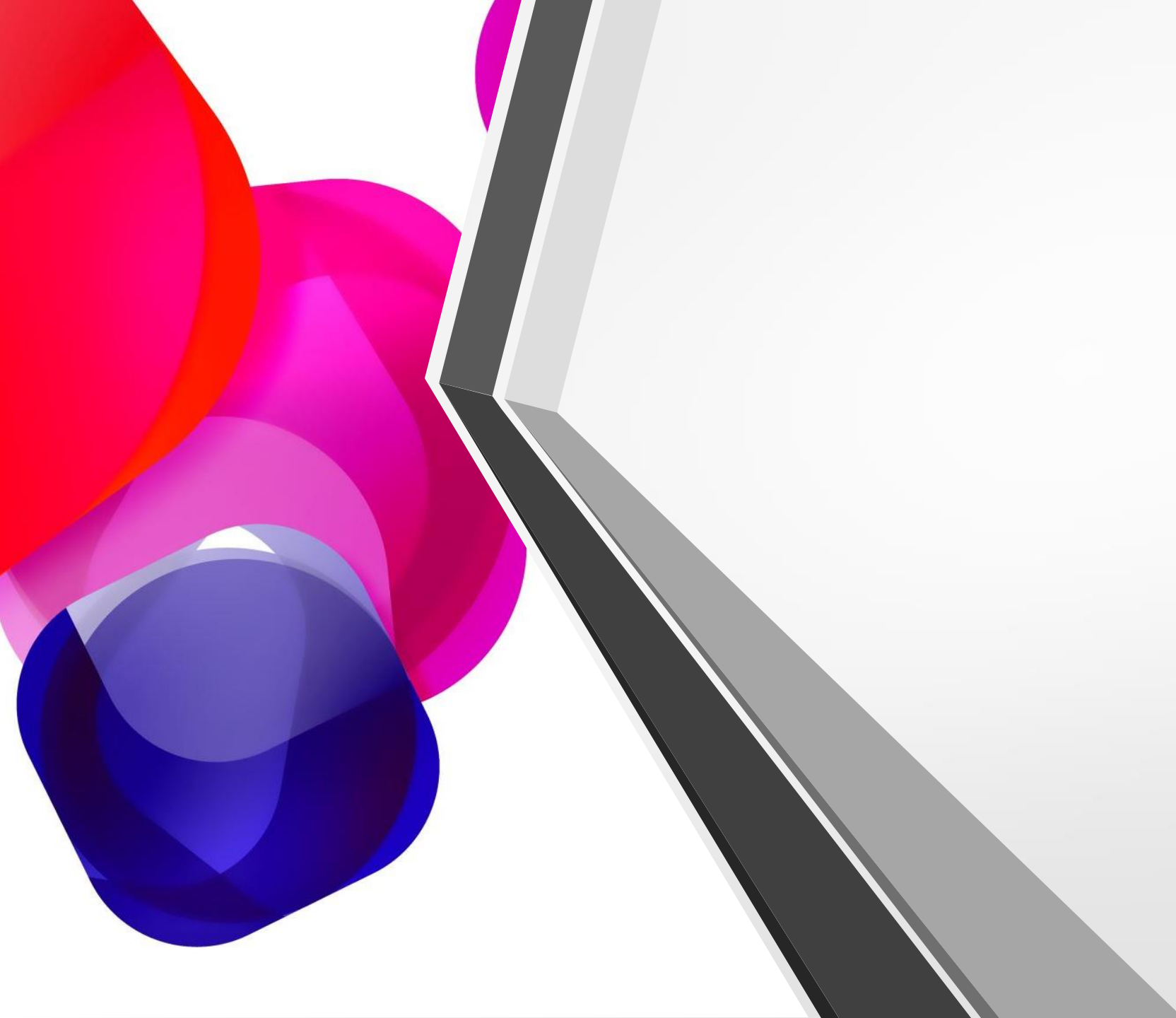
book4 = Book("Sapiens", "Yuval Noah Harari", "Non-Fiction")
book4.display_info()
```

```
Title: The Great Gatsby
Author: F. Scott Fitzgerald
Book Type: Fiction
Total Books: 23
```

```
Title: Sapiens
Author: Yuval Noah Harari
Book Type: Non-Fiction
Total Books: 28
```

```
Title: Sapiens
Author: Yuval Noah Harari
Book Type: Non-Fiction
Total Books: 29
```

```
Title: Sapiens
Author: Yuval Noah Harari
Book Type: Non-Fiction
Total Books: 3
```



Advanced Classes

What are Dunders?

What is inheritance?

What are Decorators?

What is a Dunder?

- Dunder or Magic Methods are the methods that have `__(something)__`
 - Commonly used for operator overloading
 - (providing the ability to override the functionality of a built in operator in user defined classes)
 - E.g. we want something in Python to do something it shouldn't normally be able to do
- We use dunder for classes too
 - `__init__`

```
class Book:
    total_books = 0
    book_types = ['Fiction', 'Non-Fiction']

    def __init__(self, title, author, book_type):
        self.title = title
        self.author = author
        self.book_type = book_type

        # Update class variables
        Book.total_books += 1

    def display_info(self):
        print(f>Title: {self.title}")
        print(f"Author: {self.author}")
        print(f"Book Type: {self.book_type}")
        print(f"Total Books: {Book.total_books}\n")
```

What if we put a class within another class?!

- You have thus produced the idea of Child classes!
 - This is conceptually called inheritance
 - This is different from inner/outer classes!!!
- What does this do?
 - We inherit a copy of the class
 - Do we inherit both class AND instance variables?

```
class ColorBook(Book):  
    def __init__(self, name, writer, word_length, color, has_image):  
        Book.__init__(self, name, writer, word_length)  
        self.color = color  
        self.has_image = has_image
```

What to know about Child Classes

- In this example, what are we doing?
 - Why are there two `__init__`'s?
 - Would this code work?
- Do you need to put `Book` in this?

```
class ColorBook(Book):  
    def __init__(self, name, writer, word_length, color, has_image):  
        Book.__init__(self, name, writer, word_length)  
        self.color = color  
        self.has_image = has_image
```


What to know about Child Classes

- In this example, what are we doing?
 - Why are there two `__init__`'s?
 - Would this code work?
- Do you need to put `Book` in this?
 - `Super()`
 - Why use this?
 - We don't need to include `self` with it

```
class ColorBook(Book):  
    def __init__(self, name, writer, word_length, color, has_image):  
        Book.__init__(self, name, writer, word_length)  
        self.color = color  
        self.has_image = has_image
```

```
class ColorBook(Book):  
    def __init__(self, name, writer, word_length, color, has_image):  
        super().__init__(name, writer, word_length)  
        self.color = color  
        self.has_image = has_image
```


What is a Method vs a Function?

Function

- are independent blocks of code
- Perform specific tasks
- NOT IN A CLASS

Method

- Define behaviour of the python object
- Are used for readability, reusability, and easy for debugging
- IN A CLASS

So are functions and methods the same?


- So why do we differentiate between them?!
 - OOP kind of requires it
 - Also in higher level coding it's a notable difference



What is a decorator??

- These are used in wrapping functions and classes with additional code blocks that otherwise aren't modified
- Why are they useful?
 - They add modularity
 - They add explicit/unique behaviour
- What are they used for?
 - Functional addons
 - Reusing code in other functions/classes
 - Data sanitization
 - When a variable/method can be affected by other functions, a decorator can ignore this and return the intended output
 - Function registration
 - Allows for multiple subsystems to communicate with each other

Are there built in decorators?

- 
- `@classmethod`
 - Creates the method again without creating a new instance
 - What is this doing?
 - What will it return?

Python


```
class Pizza:
    def __init__(self, ingredients):
        self.ingredients = ingredients

    def __repr__(self):
        return f'Pizza({self.ingredients!r})'

    @classmethod
    def margherita(cls):
        return cls(['mozzarella', 'tomatoes'])

    @classmethod
    def prosciutto(cls):
        return cls(['mozzarella', 'tomatoes', 'ham'])
```

Are there built in decorators?

-  @staticmethod
 - Basically an instance method but it is bound to a class rather than the objects of that class
- What does that mean?!
 - We can call it without an object for that class!
- What does that mean?
 - Don't worry about it, they're mainly used for Java within Python

```
Python

import math

class Pizza:
    def __init__(self, radius, ingredients):
        self.radius = radius
        self.ingredients = ingredients

    def __repr__(self):
        return (f'Pizza({self.radius!r}, '
                f'{self.ingredients!r})')

    def area(self):
        return self.circle_area(self.radius)

    @staticmethod
    def circle_area(r):
        return r ** 2 * math.pi
```

Are there built in decorators?

- @property



- Getter
 - Usually just referenced as @property
- Setter
 - @Variable.setter
- Deleter
 - @Variable.setter

<https://www.youtube.com/watch?v=jCzTgXFZ5bw>

```
class House:

    def __init__(self, price):
        self._price = price

    @property
    def price(self):
        return self._price

    @price.setter
    def price(self, new_price):
        if new_price > 0 and isinstance(new_price, float):
            self._price = new_price
        else:
            print("Please enter a valid price")

    @price.deleter
    def price(self):
        del self._price
```

Are there built in decorators?

- When you see “_price”
 - That means we don't touch it directly
- This is used in other programming languages mainly
- Using @property is mainly to treat a method like a variable
- @var.setter and @var.deleter will allow you to edit variables in ways we won't discuss today

```
@property
def email(self):
    return '{}.{}@email.com'.format(self.first, self.last)

@property
def fullname(self):
    return '{} {}'.format(self.first, self.last)

@fullname.setter
def fullname(self, name):
    first, last = name.split(' ')
    self.first = first
    self.last = last
```

Are there built in decorators?

- @(fxn/method names)
 - What do you think this does?
 - What do you think it outputs?

```
def makeupper(func):  
    # Function to make message Upper Case  
    def wrapper(msg):  
        uppermsg = msg  
        uppermsg = uppermsg.upper()  
        print(f"makeupper decorator conversion -> {uppermsg}")  
        return func(uppermsg)  
  
    return wrapper  
  
def maketitle(func):  
    # Function to make message Title Case  
    def wrapper(msg):  
        titlemsg = msg  
        titlemsg = msg.title()  
        print(f"maketitle decorator conversion -> {titlemsg}")  
        return func(titlemsg)  
  
    return wrapper  
  
# Assign Multiple Decorators  
@makeupper  
@maketitle  
def printgreeting(message):  
    print(f"Final message -> {message}")  
  
# Call the Function  
printgreeting("hello world")
```


Are there built in decorators?

- @(fxn/method names)
 - What do you think this does?
 - What do you think it outputs?

```
maketitle decorator conversion -> Hello World  
makeupper decorator conversion -> HELLO WORLD  
Final message -> HELLO WORLD
```

```
def makeupper(func):  
    # Function to make message Upper Case  
    def wrapper(msg):  
        uppermsg = msg  
        uppermsg = uppermsg.upper()  
        print(f"makeupper decorator conversion -> {uppermsg}")  
        return func(uppermsg)  
  
    return wrapper  
  
def maketitle(func):  
    # Function to make message Title Case  
    def wrapper(msg):  
        titlemsg = msg  
        titlemsg = msg.title()  
        print(f"maketitle decorator conversion -> {titlemsg}")  
        return func(titlemsg)  
  
    return wrapper  
  
# Assign Multiple Decorators  
@makeupper  
@maketitle  
def printgreeting(message):  
    print(f"Final message -> {message}")  
  
# Call the Function  
printgreeting("hello world")
```

"Oh yeah I never use
classes in my code..."

In Python

In Java

