PROGRAMMING: OBJECT-ORIENTED APPROACH 🧑‍💻

CLASSES

- Press `Space` to navigate through the slides
- Use `Shift+Space` to go back
- Save as **PDF**:
  - Open **Chrome** then **click here**
  - Press **Ctrl+P/Cmd+P** to print
    - Destination: **Save as PDF**
    - Layout: **Landscape**
  - Press **Save** button

BEFORE JUMPING INTO CLASSES

Let's review a couple of new things in **Python** that will help us learning classes today:

- **Format Strings**
- Working with **dates**

FORMAT STRINGS

- Format strings are a way to add variables into strings in python.
- They are incredibly useful in all sorts of applications, like greeting a user with their name when they login to an app, or writing a preformatted message to tell people the temperature etc.
- Basically anywhere that you would normally want to print a variable, format strings are usually the best option.
- They come in a few formats, but I'm only going to show you the 'recommended one' (see end of this section for details about others).

FORMAT STRINGS

- To use format strings simply add an f before your quotes (double or single) to declare a string, and where you want a variable to appear add it in between curly braces.
- For example, let's say you wanted to greet someone when they login, the code could look like this:

```
name = "John Doe"

greeting = f"Welcome, {name}!"

print(greeting) # Prints: Welcome John Doe!
```

FORMAT STRINGS

- One thing to remember with these strings is that they are **created on call**, meaning they only update when they are created and not when the variable is updated.
- For example:

```
name = "John Doe"

greeting = f"Welcome, {name}!"

name = "Kieran Wood"

print(greeting) # Prints: Welcome John Doe!

greeting = f"Welcome, {name}!" # Since it's recreated it picks up the new value o

print(greeting) # Prints: Welcome Kieran Wood!
```

DATES

- Python comes with a great module for handling dates and times called **datetime**.
- Let's look into some examples of basic usage of the **datetime** module:
  - Create Date object
  - Compare date objects
  - Get date object attributes
  - Get Current Date

CREATE DATE OBJECT

- The module works off of classes that you can use to create objects, to create a simple date object you just need to provide 3 attributes – a **year (int)**, a **month (int)**, and **day (int)**:

```
import datetime

appolo_11_launch = datetime.date(1959, 9, 13)
```

COMPARE DATE OBJECTS

- You can make comparisons to date objects the same way you would regular numbers:

```
import datetime

appolo_11_launch = datetime.date(1959, 9, 13)

falcon_9_first_launch = datetime.date(2010, 6, 4)

print(falcon_9_first_launch > appolo_11_launch) # prints: True
```

GET DATE OBJECT ATTRIBUTES

- You can get the attributes of a date object (the year, month, day) the same way you access class attributes:

```
import datetime

appolo_11_launch = datetime.date(1959, 9, 13)

appolo_11_launch.year # prints: 1959

appolo_11_launch.month # Prints: 9

appolo_11_launch.day # Prints: 13
```

GET CURRENT DATE

- You can get the current date using the datetime.date.today() function, which returns a datetime object of today's date:

```
import datetime

current_datetime = datetime.date.today() # Returns datetime object of todays date
```

CLASSES

- **Functions** *do* specific things, classes *are* specific things.
- **Classes** often have **methods**, which are functions that are associated with a particular class, and do things associated with the thing that the class is
  - If all you want is to do something, a **function** is all you need
  - Don't create a class that only has two methods and one of them is **init**
- Essentially, a **class** is a way of grouping functions (as **methods**) and data (as **properties**) into a logical unit revolving around a certain kind of thing
  - If you don't need that grouping, there's no need to make a class.

CLASSES

- **Classes** are a way of bundling data (attributes) and functions (sometimes called methods) into abstractions in Python.
- For example, let's say you are writing an app to store a bunch of data about animals.
- For this you can set up a class that has the basic attributes of all animals like species name, endemic regions (where it lives), common name (what people know it as) etc.

- The python code for this would look like:

```
class Animal:
  def __init__(self, species_name, region
    """A class to represent a generic ani

    Attributes
    ----------
    species_name : (str)
        The technical species name of the
    regions : (list[str])
        A list of regions the animal is e
    common_name : (str)
        The colloquial name of the animal
    """
    self.species_name = species_name
    self.regions = regions
    self.common_name = common_name
```

CLASSES

- So we can break down the example in a moment, first let's start with what you can now do with the above class.
- Once you have a class, it can act as a template to create *instances*.
- You can think of this with the analogy that a *class* is a cookie cutter, and an *instance* is a cookie that has been cut from the cutter.
- Let's use our *Animal class* (cookie cutter) to create an *instance* of a *Common Leopard Gecko*:

```
leopard_gecko = Animal("Eublepharis macularius",
    ["Afghanistan","Pakistan","India", "Iran"],
    "Common Leopard Gecko")
```

CLASSES

- Now that we have have *instantiated* the leopard gecko instance, we can use a simple syntax (`instance.variable_name`) to access the variables.
- For example if we wanted to know the common name of a *Common Leopard Gecko* we could use:

```
leopard_gecko.common_name # Returns 'Common Leopard Gecko'
```

CLASSES

- We can also add methods (functions) specific to the class, for example let's write a method that prints info about the animal:

```
class Animal:
  def __init__(self, species_name, regions, common_name):
    """A class to represent a generic animal

    Attributes
    ----------
    species_name : (str)
        The technical species name of the animal
    regions : (list[str])
        A list of regions the animal is endemic to
    common_name : (str)
        The colloquial name of the animal
    """
    self.species_name = species_name
    self.regions = regions
    self.common_name = common_name
  def print_info(self):
    """Prints information about animal instance"""
    print(f"\nCommon Name: {self.common_name}\nSpecies: {self.species_name}\nRegi
```

CLASSES

- Now we can take the leopard gecko example from earlier and use our method using the syntax (instance.method()):

```
leopard_gecko = Animal("Eublepharis macularius", ["Afghanistan","Pakistan","India"

"""Prints (not returns)
Common Name: Common Leopard Gecko
Species: Eublepharis macularius
Regions: ['Afghanistan','Pakistan','India', 'Iran']
"""
leopard_gecko.print_info()
```

CLASSES

- There was a lot going on in the last example so lets break down what happened.
- First we start by *defining* our class (usually called the *class definition*) with:

```
class Classname: # Notice for classes the convention is to start them with a capit
  def __init__(self): # This method will be explained later on
    pass
```

- Following our *class definition* right away we define a **`__init__`** method.
- The **`__init__`** method is explained in further detail below.
- But first let's take a look at that funny *self* that we've been putting in front of our variables.

CLASSES

SELF; INSTANCE VS CLASS ATTRIBUTES

- As you can see, for all of the *instance attributes* (variables specific to each instance and not the overall class) we are adding a *self* in front with a dot.
- This is because when we create an *instance*, all of the variables are *localised* to that *instance*.
- The word 'self' is used to represent the instance of a class and by using the "self" keyword we access the attributes and methods of the class in python.
- So for example if we use the animal class from earlier to create two different animal instances, the variables don't overlap.

```
leopard_gecko = Animal("Eublepharis macularius",
    ["Afghanistan","Pakistan","India", "Iran"],
    "Common Leopard Gecko")

arctic_fox = Animal("Vulpes lagopus",
    ["The Arctic"],
    "Arctic fox")

leopard_gecko.common_name # Returns 'Common Leopard Gecko'

arctic_fox.common_name # Returns 'Arctic fox'
```

CLASSES

SELF; INSTANCE VS CLASS ATTRIBUTES

- This is the same with *instance methods*.
- The variables it pulls are specific to the *instance* and not the *class*.

```
leopard_gecko = Animal("Eublepharis macularius",
    ["Afghanistan","Pakistan","India", "Iran"],
    "Common Leopard Gecko")

arctic_fox = Animal("Vulpes lagopus",
    ["The Arctic"],
    "Arctic fox")

leopard_gecko.print_info()
"""Prints (not returns)
'Common Name: Common Leopard Gecko
Species: Eublepharis macularius
Regions: ["Afghanistan","Pakistan","India", "Iran"]'
"""

arctic_fox.print_info()
"""Prints (not returns)
'Common Name: Arctic fox
Species: Vulpes lagopus
Regions: ["The Arctic"]'
```

CLASSES

SELF; INSTANCE VS CLASS ATTRIBUTES

- *Class attributes* on the other hand are attributes that are common among **all** *instances* of a *class*.
- For example: let's say you wanted to keep a counter that goes up by 1 for every time a new animal is added.
- Since this information, is not specific to an *instance*, but rather to every instance of a given *class* you would want it to be accessible to every instance.
- The code to do something like this would be...

```
class Animal:
  counter = 0 # Initialize counter to 0
  # This ^^ is a class variable since it
  def __init__(self, species_name, regio
    """A class to represent a generic an

   Attributes
    ----------
    species_name : (str)
        The technical species name of th
    regions : (list[str])
        A list of regions the animal is
    common_name : (str)
        The colloquial name of the anima
    """
    self.species_name = species_name
    self.regions = regions
    self.common_name = common_name
    Animal.counter += 1 # Accessing and
```

CLASSES

SELF; INSTANCE VS CLASS ATTRIBUTES

- Now there is a counter variable that can be used to find out how many animals have been instantiated

```
print(Animal.counter) # Prints 0; since no Animal's have been instantiated
leopard_gecko = Animal("Eublepharis macularius",
    ["Afghanistan","Pakistan","India", "Iran"],
    "Common Leopard Gecko")

print(Animal.counter) # Prints 1; since the Leopard Gecko has been instantiated

arctic_fox = Animal("Vulpes lagopus",
    ["The Arctic"],
    "Arctic fox")

print(Animal.counter) # Prints 2; since the Leopard Gecko, and Arctic fox have bee

# Both below calls print 2; Class variables are also accessible in any instance
print(arctic_fox.counter)
print(leopard_gecko.counter)
```

- As you can see because the variable belongs to the *class* and not the *instance*, it is available to both the class as a variable, or any *instances* of the *Animal* class.

CLASSES

CLASS METHODS

- Methods are functions that are accessible through class *instances*, for example let's say you want to create a function to print all of the *attributes of a class* you could define the function in the class and then use the *self* operator to print the information.

- We have already seen this in fact in the above examples with our Animal class, the print_info() method used earlier is a class method.

CLASSES

CLASS METHODS: BASIC SYNTAX

- Setting up class methods is the same as setting up a regular function, you just need to indent it to the same line as the class and **always** pass *self* as an argument. For example:

```
class Animal:
  def __init__(self, species_name, regions, common_name):
    """A class to represent a generic animal

    Attributes
    ----------
    species_name : (str)
        The technical species name of the animal
    regions : (list[str])
        A list of regions the animal is endemic to
    common_name : (str)
        The colloquial name of the animal
    """
    self.species_name = species_name
    self.regions = regions
    self.common_name = common_name
  def print_info(self):
    """Prints information about animal instance"""
    print(f"\nCommon Name: {self.common_name}\nSpecies: {self.species_name}\nRegi
```

CLASSES

DUNDER OR MAGIC METHODS

- Dunder or magic methods in Python are the methods having two prefix and suffix underscores in the method name.
- Dunder here means “Double Under (Underscores)”.
- These are commonly used for operator overloading – we will cover this later in the semester.
- Few examples for magic methods are: **init**, **add**, **len**, **repr** etc.

`__init__` METHOD

- The `__init__` is a reserved method in python acts as a constructor (sort of) in Python.
- This method is called when an object is created from a class and it allows the class to initialise the attributes of the class.
- This means that it 'constructs' the instance.

CLASSES

`__init__` METHOD

- In our analogy of a cookie cutter from earlier, the `__init__` method would be the actual cutting of the cookie.
- The method is run every time you *instantiate* an instance. For example when you run the leopard_gecko example from before:

```
leopard_gecko = Animal("Eublepharis macularius",
    ["Afghanistan","Pakistan","India", "Iran"],
    "Common Leopard Gecko")
```

The variable is making an *implicit* call to `__init__`, this would roughly be equivalent to:

```
leopard_gecko = Animal.__init__("Eublepharis macularius",
    ["Afghanistan","Pakistan","India", "Iran"],
    "Common Leopard Gecko")
```

CLASSES

__init__ METHOD

- We can see this more clearly by switching examples a bit. Let's take the following class:

```
class CookieBaker:
  def __init__(self, number_of_cookies):
    """ Example class that is used to show off the __init__ method.
    The __init__ method calls prints 'Cookie Baked' as many times as there are nur

    Attributes
    ----------
    number_of_cookies(int): How many cookies to bake
    """
    print(f"__init__ method called, creating {number_of_cookies} cookie(s):")
    self.number_of_cookies = number_of_cookies
    for cookie in range(number_of_cookies):
      print("Cookie Baked!")
```

CLASSES

`__init__` METHOD

- As you can see, you can do some basic logic in the **`__init__`** method, such as for loops.
- You can also call methods, just make sure to include **`self.`** when calling them since you are inside the instance.

```
class CookieBaker:
  def __init__(self, number_of_cookies):
    """ Example class that is used to show off the __init__ method.
    The __init__ method calls the bake_cookie() method as many times as there are

    Attributes
    ----------
    number_of_cookies(int): How many cookies to bake
    """
    print(f"__init__ method called, creating {number_of_cookies} cookie(s):")
    self.number_of_cookies = number_of_cookies
    for cookie in range(number_of_cookies):
      self.bake_cookie()

  def bake_cookie(self):
    """Print's 'Cookie Baked!'."""
    print("Cookie Baked!")
```

ADDITIONAL INFO

CLASS ATTRIBUTES: ACCESS

- Keep in mind that like other variables python will let you override class variables **without question**.
- So they can be modified from the *class* at will.
- For example:

```
print(Animal.counter) # Prints 0; since no Animal's have been instantiated
leopard_gecko = Animal("Eublepharis macularius",
    ["Afghanistan","Pakistan","India", "Iran"],
    "Common Leopard Gecko")

print(Animal.counter) # Prints 1; since the Leopard Gecko has been instantiated

Animal.counter = 35 # Overriding the variable from the class

print(Animal.counter) # Prints 35; since the attribute has been overridden
print(leopard_gecko.counter) # Prints 35; since the attribute has been overridden
```

ADDITIONAL INFO

CLASS ATTRIBUTES: ACCESS

- **But** if you try to modify a class variable from an instance, then it will create an *instance* variable that is now local to the instance while leaving the class variable in tact:

```
print(Animal.counter) # Prints 0; since no Animal's have been instantiated
leopard_gecko = Animal("Eublepharis macularius",
    ["Afghanistan","Pakistan","India", "Iran"],
    "Common Leopard Gecko")

print(Animal.counter) # Prints 1; since the Leopard Gecko has been instantiated

Animal.counter = 35 # Overriding the variable from the class

print(Animal.counter) # Prints 35; since the attribute has been overridden

leopard_gecko.counter = 26 # creating an instance variable from the class attribut

print(Animal.counter) # Prints 35; since the class attribute WONT be modified by
print(leopard_gecko.counter) # Prints 26; since the instance attribute has been c
```

ADDITIONAL INFO

DATACLASSES IN PYTHON

- If you are just storing variables there is also a useful module inside the Python standard library called dataclasses this library makes creating useful classes that just store data much faster.

- Let's take a look at this example:

```
import datetime
class user:
  def __init__(self, name, age, sign_up_date, birthday, premium_member):
    """A class to represent a generic animal

    Attributes
    ----------
    name(str): The technical species name of the animal
    age(str): A list of regions the animal is endemic to
    sign_up_date(datetime.datetime): A datetime object of the day the user signed
    birthday(datetime.datetime): A datetime object of the users birthday
    premium_member(bool): Whether the user is on premium or free subscription
    """
    self.name = name
    self.age = age
    self.sign_up_date = sign_up_date
    self.birthday = birthday
    self.premium_member = premium_member
```

ADDITIONAL INFO

DATACLASSES IN PYTHON

- Now thats a lot of *self.attribute_name*'s, dataclasses will automate the `__init__` method (and `__repr__` method).

```
import datetime
from dataclasses import dataclass

@dataclass
class user:
    """A class to represent a generic animal

    Attributes
    ----------
    name(str): The technical species name of the animal
    age(str): A list of regions the animal is endemic to
    sign_up_date(datetime.datetime): A datetime object of the day the user signed
    birthday(datetime.datetime): A datetime object of the users birthday
    premium_member(bool): Whether the user is on premium or free subscription
    """
    name:str
    age:str
    sign_up_date:datetime.datetime
    birthday:datetime.datetime
```

ADDITIONAL INFO

GLOSSARY

- ***Class***: A template to create *Instance(s)/Object(s)* from.
    - Classes exist to bundle data (attributes) and functions (methods) into abstractions that are meaningful.
    - A good analogy is to think of a cookie cutter as a class, that is a template used to cut (*instantiate*) cookies (*Instance(s)/Object(s)*)
- For example you could have an Animal class that can be used to create *Instance(s)/Object(s)* to bundle data and functions about animals, or a user class that can be used to create *Instance(s)/Object(s)* to bundle data and functions about each user of an app.

ADDITIONAL INFO
GLOSSARY

- ***Instance/Object***. An object representing something, created from a class (used as a template).
    - A good analogy is to think of a cookie cutter as a class, that is a template used to cut (*instantiate*) cookies (*Instance(s)/Object(s)*).
- For example if you had an Animal class, you could use it to *instantiate* a leopard gecko instance that has all the data (attributes) and functions (methods) necessary to represent a leopard gecko.
- In Python people use *Instance* and *Object* interchangeably, and the same is true for many other object-oriented languages.

ADDITIONAL INFO

GLOSSARY

- ***Instantiate***: The act of creating (initialising) an *Instance/Object* from a *class*.
    - A good analogy is to think of a cookie cutter as a *class*, that is a template used to cut (*instantiate*) cookies (*Instance(s)/Object(s)*)
- ***Attribute***: A variable that is specific to a *class* or *Instance/Object*.
- ***Method***: A function that is specific to a *class* or *Instance/Object*.
- ***Constructor***: What typically gets called on *instantiation* of an *Instance/Object*.
    - This is a concept used broadly in object-oriented languages, but in python this roughly corresponds to the `__init__` method.