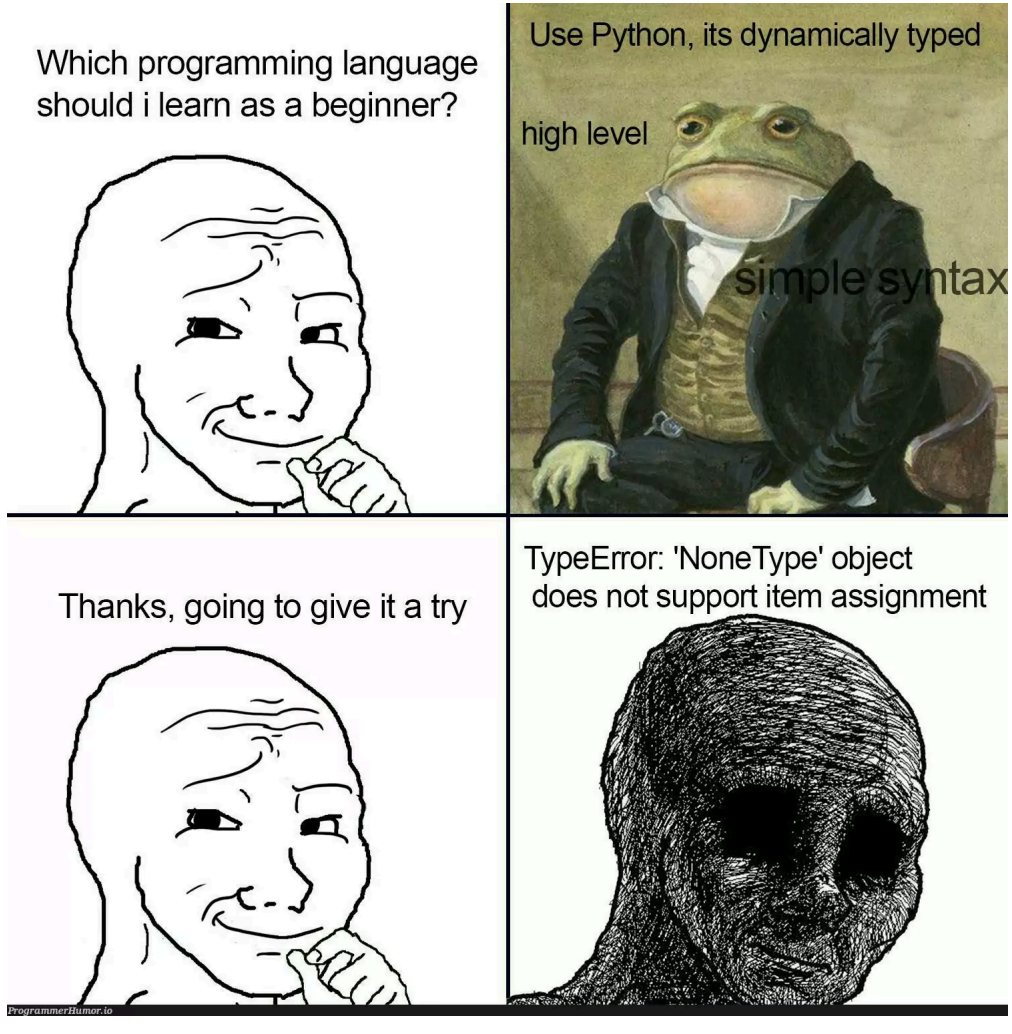
# Discussion 9

DSC 20, Winter 2024

# Meme of the Week



# Agenda

- is vs ==
- Shallow vs Deep copies
- __str__ and __repr__
- special methods
- Exceptions
- KNN
- f strings

# is vs ==

**is**: The `is` operator checks whether two variables point to the same object in memory, i.e., it compares the memory addresses of the objects. It evaluates to True if the operands point to the same object and False otherwise. It is used to check object identity.

**==**: The `==` operator, on the other hand, compares the values of two objects for equality. It evaluates to True if the objects referred to by the variables are equivalent (i.e., have the same content), regardless of whether they are the same object in memory. This operator is used to compare the contents or data of two objects.

# Shallow vs Deep Copies in Python

**Shallow Copies**: A shallow copy copies the references to a given object. This means that if the original object contains any references to mutable objects, the duplicate object will reference the same objects, not copies of them.

**Deep Copies**: In contrast, a deep copy creates a new object of the original. Consequently, the copied object is fully independent of the original, and changes to it do not affect the original object or vice versa.

pythontutor

In [2]:
```
# shallow copy
first_lst = [2,4,6]
second_lst = first_lst
# deep copy
third_lst = list(first_lst)

print(first_lst==second_lst)
print(first_lst is second_lst)
print(first_lst==third_lst)
print(first_lst is third_lst)
```

```
True
True
True
False
```

# Special Methods

- Unique class methods that impose specific functionality
- Denoted by __ prepending and appending method names
- The one you're most familiar with is __init__
- Also called Dunder Methods!

## __str__

- human readable representation of a class
- invoked when the string representation of a class is necessary (i.e. in a print or casted to a string)

In [9]:
```
class phone:
    def __init__(self, name, owner, memory):
        self.name = name
        self.owner = owner
        self.memory = memory

    def __str__(self):
        return f'{self.name} owned by {self.owner}'
```

In [10]:
```
iphone = phone('iPhoneIX', 'nikki',800)
print(f'string representation of phone: {iphone}')
```

```
string representation of phone: iPhoneIX owned by nikki
```

# Checkpoint

Given the following code, what is the result of the final expression?

In [ ]:
```
class phone:
    def __init__(self, name, owner, memory):
        self.name = name
        self.owner = owner
        self.memory = memory

    def __str__(self):
        return f'{self.name} owned by {self.owner}'
iphone = phone('iPhoneIX', 'nikki',800)
print(f'computational representation of phone: {iphone.__repr__()}')
```

A. computational representation of phone: iPhoneIX owned by nikki

B. computational representation of phone:

C. computational representation of phone: <**main**.phone object at 0x7fc2c8d54670>

D. Error

## Checkpoint Solution

In [13]:
```
print(f'computational representation of phone: {iphone.__repr__()}')
print('ew!')
```

```
computational representation of phone: <__main__.phone object at
0x7fc2c8d54670>
ew!
```

Not exactly the ideal output. Since `__repr__` wasn't explicitly defined, python used the default `__repr__` which is the memory address of an object.

# __repr__

- computer/technical representation of a class
- commonly written so that the returned value can be used to recreate the object

In [27]:
```
class phone:
    def __init__(self, name, owner, memory):
        self.name = name
        self.owner = owner
        self.memory = memory

    def __repr__(self):
        return f'phone("{self.name}", "{self.owner}", {self.memory})'
```

In [17]:
```
iphone = phone('iPhoneIX', 'nikki',800)
print(f'computational representation of phone: {iphone.__repr__()}')
```

```
computational representation of phone: phone("iPhoneIX", "nikk
i", 800)
```

# Checkpoint

Given the following code, what is the result of the final expression?

In [ ]:
```
class phone:
    def __init__(self, name, owner, memory):
        self.name = name
        self.owner = owner
        self.memory = memory

    def __repr__(self):
        return f'phone("{self.name}", "{self.owner}", {self.memory})'
iphone = phone('iPhoneIX', 'nikki',800)
print(f'string representation of phone: {iphone}')
```

A. string representation of phone: phone("iPhoneIX", "nikki", 800)

B. string representation of phone:

C. string representation of phone: <**main**.phone object at 0x7fc2c8d54670>

D. Error

## Checkpoint Solution

In [20]:
```
iphone = phone('iPhoneIX', 'nikki',800)
print(f'string representation of phone: {iphone}')
```

```
string representation of phone: phone("iPhoneIX", "nikki", 800)
```

Although `__str__` is not explicitly defined here, since `__repr__` is defined, Python will derive the string representation using `__repr__`.

# __repr__ and __str__

In [29]:
```
class phone:
    def __init__(self, name, owner, memory):
        self.name = name
        self.owner = owner
        self.memory = memory

    def __str__(self):
        return f'{self.name} owned by {self.owner}'

    def __repr__(self):
        return f'phone("{self.name}", "{self.owner}", {self.memory})'
```

In [30]:
```
iphone = phone('iPhoneIX', 'nikki',800)
print('string representation of phone: ' + str(iphone))
print('computational representation of phone: ' + repr(iphone))
```

```
string representation of phone: iPhoneIX owned by nikki
computational representation of phone: phone("iPhoneIX", "nikk
i", 800)
```

In [31]:
```
iphone_copy = eval(iphone.__repr__()) # created a copy using eval on th
print('Are they the same object: ' + str(iphone is iphone_copy))
print('Are they the same type: ' + str(type(iphone) == type(iphone_copy
print('Same string and repr represention: '+ \
    f'{str(iphone) == str(iphone_copy) and repr(iphone)==repr(iphone_co
```

```
Are they the same object: False
Are they the same type: True
Same string and repr represention: True
```

In [32]:
```
iphone_copy2 = eval(iphone.__str__())
```

```
Traceback (most recent call last):

  File ~/anaconda3/lib/python3.10/site-packages/IPython/core/int
eractiveshell.py:3460 in run_code
    exec(code_obj, self.user_global_ns, self.user_ns)

  Cell In[32], line 1
    iphone_copy2 = eval(iphone.__str__())

  File <string>:1
    iPhoneIX owned by nikki
             ^
SyntaxError: invalid syntax
```

## __(lt, gt, eq, etc.)__

other dunder methods encode certain behaviors of the class, usually in accordance with each other.

In [22]:
```
class phone:
    def __init__(self, name, owner, memory):
        self.name = name
        self.owner = owner
        self.memory = memory

    def __repr__(self):
        return f'phone("{self.name}", "{self.owner}", {self.memory})'

    def __lt__(self, other_phone):
        return self.memory < other_phone.memory

    def __eq__(self, other_phone):
        return self.name == self.name
```

In [23]:
```
iphone = phone('iPhoneIX', 'nikki',800)
samsung = phone('SamsungEarth', 'nikki',1600)
print(f'is the iphone less than the samsung (memory): {iphone<samsung}'
print(f'is the iphone greater than the samsung (memory): {iphone>samsur
print(f'are the two phones the same (name): {iphone==samsung}')
```

```
is the iphone less than the samsung (memory): True
is the iphone greater than the samsung (memory): False
```

```
are the two phones the same (name): True
```

# Checkpoint

Given the following code, what is the result of the final expression?

In [25]:
```
class phone:
    def __init__(self, name, owner, memory):
        self.name = name
        self.owner = owner
        self.memory = memory

    def __repr__(self):
        return f'phone("{self.name}", "{self.owner}", {self.memory})'

    def __lt__(self, other_phone):
        return self.memory < other_phone.memory

    def __eq__(self, other_phone):
        return self.name == self.name
```

In [ ]:
```
iphone_1 = phone('iPhoneIX', 'nikki',800)
iphone_2 = phone('iPhoneIX', 'ben',1600)
print(iphone_1 == iphone_2, iphone_1 is iphone_2)
```

A. True True

B. True False

C. False True

D. True True

## Checkpoint Solution

In [26]:
```
iphone_1 = phone('iPhoneIX', 'nikki',800)
iphone_2 = phone('iPhoneIX', 'ben',1600)
print(iphone_1 == iphone_2, iphone_1 is iphone_2)
```

```
True False
```

# Exceptions

- Python class that represents Errors
- Triggered by any event that disrupts normal program flow
- utilizes `raise` keyword

# Common Exception Types and Cases

- **KeyError** - related to dictionaries; attempted access using a key not present in the object
- **IndexError** - related to lists/strings; attempted acccess of an index that's out of range
- **TypeError** - attempt to unify non - matching data types (ex. str + int) or attempt to access unknown attribute of datatype
- **FileNotFoundError** - related to files; attempted to open a file name that can't be found

# raise

python keyword to throw an error.

**syntax**: raise [exception class](error message // optional)

In [49]:
```
def bar_entry(age):
    if age < 21:
        raise ValueError("too young to drink...")
bar_entry(18)
```

```
---------------------------------------------------------------------------
ValueError                                Traceback (most recent
call last)
Cell In[49], line 4
      2     if age < 21:
      3         raise ValueError("too young to drink...")
----> 4 bar_entry(18)

Cell In[49], line 3, in bar_entry(age)
      1 def bar_entry(age):
      2     if age < 21:
----> 3         raise ValueError("too young to drink...")

ValueError: too young to drink...
```

# try except else finally

- Exceptions can be handled in `try-except` blocks, preventing code from terminating the moment an error happens.
- `Except` generally handles specific error classes (ex. except `zerodivisionerror`)
- just like with files, it's common to specify a handle for easier processing (`as e`)
- `else` is code that's ran in the case of no errors occurring
- `finally` is code that will always run

# Checkpoint

Given the following code, what is the result of the final expression?

In [ ]:
```
def attempt_bar_entry(age):
    try:
        bar_entry(age)
    except ValueError as e:
        print(e)
    else:
        print('welcome in!')
    finally:
        print('-----------------------------------')

attempt_bar_entry(18)
attempt_bar_entry(21)
```

A.

```
too young to drink...

-----------------------------------

welcome in!

-----------------------------------
```

B.

```
welcome in!
```

```
----------------------------------------
```

```
welcome in!
```

```
----------------------------------------
```

C.

```
too young to drink...
```

```
welcome in!
```

D. Error, Error

# Checkpoint Solution

In [63]:
```
attempt_bar_entry(18)
attempt_bar_entry(21)
```

```
too young to drink...
------------------------------------
welcome in!
------------------------------------
```

# try-except vs assert

Assert statements are used to validate inputs and prevent logical errors. Try-except is used to catch error generating inputs / code.

In [35]:

```python
def process_file(filepath):
    # What happens if I pass in an incorrect filepath?
    assert isinstance(filepath, str) # asserts can't catch such errors
    with open(filepath, 'r') as f:
        return f.readlines()
process_file('nope')
```

```
---------------------------------------------------------------------------
FileNotFoundError                         Traceback (most recent call last)
Cell In[35], line 6
      4     with open(filepath, 'r') as f:
      5         return f.readlines()
----> 6 process_file('nope')

Cell In[35], line 4, in process_file(filepath)
      1 def process_file(filepath):
      2     # What happens if I pass in an incorrect filepath?
      3     assert isinstance(filepath, str) # asserts can't catch such errors
----> 4     with open(filepath, 'r') as f:
      5         return f.readlines()

File ~/anaconda3/lib/python3.10/site-packages/IPython/core/interactiveshell.py:282, in _modified_open(file, *args, **kwargs)
    275 if file in {0, 1, 2}:
    276     raise ValueError(
    277         f"IPython won't let you open fd={file} by default "
```

```
    278         "as it is likely to crash IPython. If you know w
hat you are doing, "
    279         "you can use builtins' open."
    280     )
--> 282 return io_open(file, *args, **kwargs)

FileNotFoundError: [Errno 2] No such file or directory: 'nope'
```

In [36]:
```
def process_file(filepath):
    # What happens if I pass in an incorrect filepath?
    try:
        f = open(filepath,'r')
    except FileNotFoundError as e:
        print('oops! filepath is not valid')
    else:
        return f.read()
process_file('nope')
print('-------------------------------------')
process_file('files/sample.txt')
```

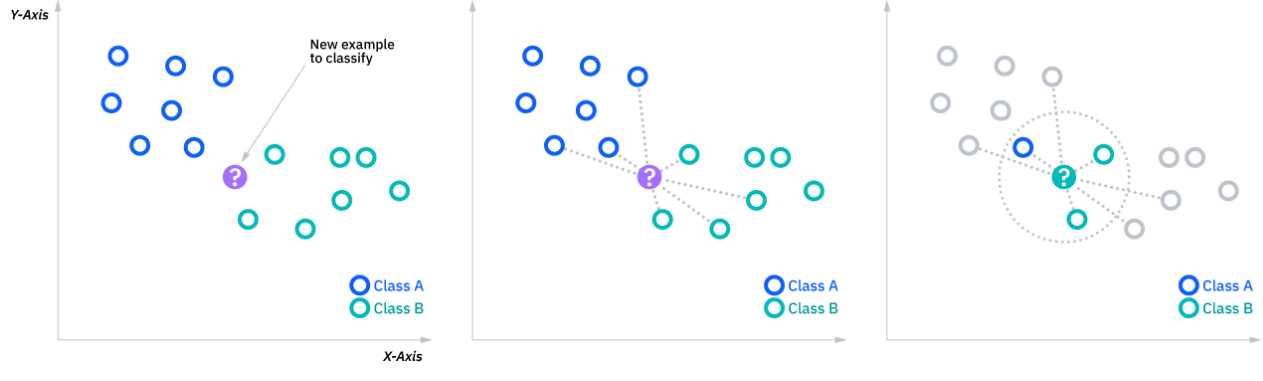
```
oops! filepath is not valid
-------------------------------------
```

Out[36]:
```
'Loki season 2 was pretty good'
```

# More Formally, KNN

**idea**: Given an unseen value of flipper_length and body_mass, how can we determine what species of penguins it belongs to?

The KNN approach is simple - if I look at the k nearest points to this new value, and I take the most common species among them, then this point is **most likely** the same species as the most common species among the neighbors.



# Procedure

**Step 1**: Given a new point *X*, quantify the distance between all points and *X*

**Step 2**: Take the **k-nearest** points to *X* into consideration

**Step 3**: Classify *X* as the most common label among the neighbors

**note:** As the name implies, the key parts of this algorithm are **k** and **nearest**.

**k** is often referred to as a **hyper-parameter**, or a value that you choose independently. There are often procedures to select a good k, but this varies depending on the context.

**Nearest** refers to quantifying distance - one such way is to use euclidian distance (distance formula), but there are many other "distances" that can be used (ex. Manhattan Distance).

In [3]:
```
# load in penguins data
data = pd.read_csv('data/penguins.csv').dropna()
data.head()
```

Out[3]:

| | species | island | culmen_length_mm | culmen_depth_mm | flipper_length_m |
|---|---|---|---|---|---|
| **0** | Adelie | Torgersen | 39.1 | 18.7 | 18 |
| **1** | Adelie | Torgersen | 39.5 | 17.4 | 186 |
| **2** | Adelie | Torgersen | 40.3 | 18.0 | 195 |
| **4** | Adelie | Torgersen | 36.7 | 19.3 | 193 |
| **5** | Adelie | Torgersen | 39.3 | 20.6 | 190 |

In [4]:
```
# get columns of interest
mapping = {'Adelie':0, 'Chinstrap':1, 'Gentoo':2}
reverse_mapping = {v:k for k,v in mapping.items()}
data = data[['species', 'flipper_length_mm', 'body_mass_g']]
data['species'] = data['species'].apply(lambda x: mapping[x])
data.head()
```

Out[4]:

| | species | flipper_length_mm | body_mass_g |
|---|---|---|---|
| **0** | 0 | 181.0 | 3750.0 |
| **1** | 0 | 186.0 | 3800.0 |
| **2** | 0 | 195.0 | 3250.0 |
| **4** | 0 | 193.0 | 3450.0 |
| **5** | 0 | 190.0 | 3650.0 |

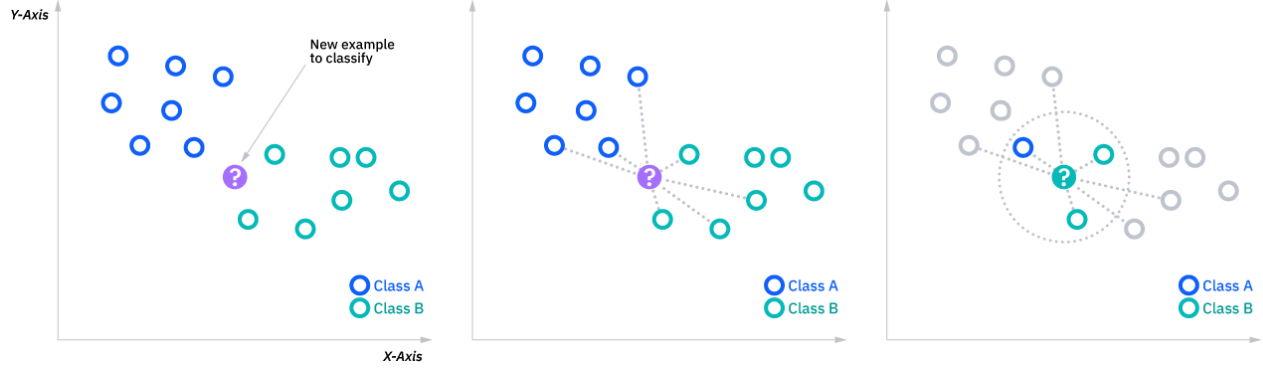
# Data Input

- It's convention to call input data X and prediction class Y
- In this example, we are using body mass and flipper length as input (X) and species as prediction (Y)

In [37]:
```
# define input and prediction
X = data[['body_mass_g', 'flipper_length_mm']]
y = data['species']
```

# fit/predict

- In order for our model to work, we need to mount the data using `fit`
- Once the data is loaded, we can call predict on other inputs using `predict`
- Note that this work differs from your project in that I am preloading a KNN

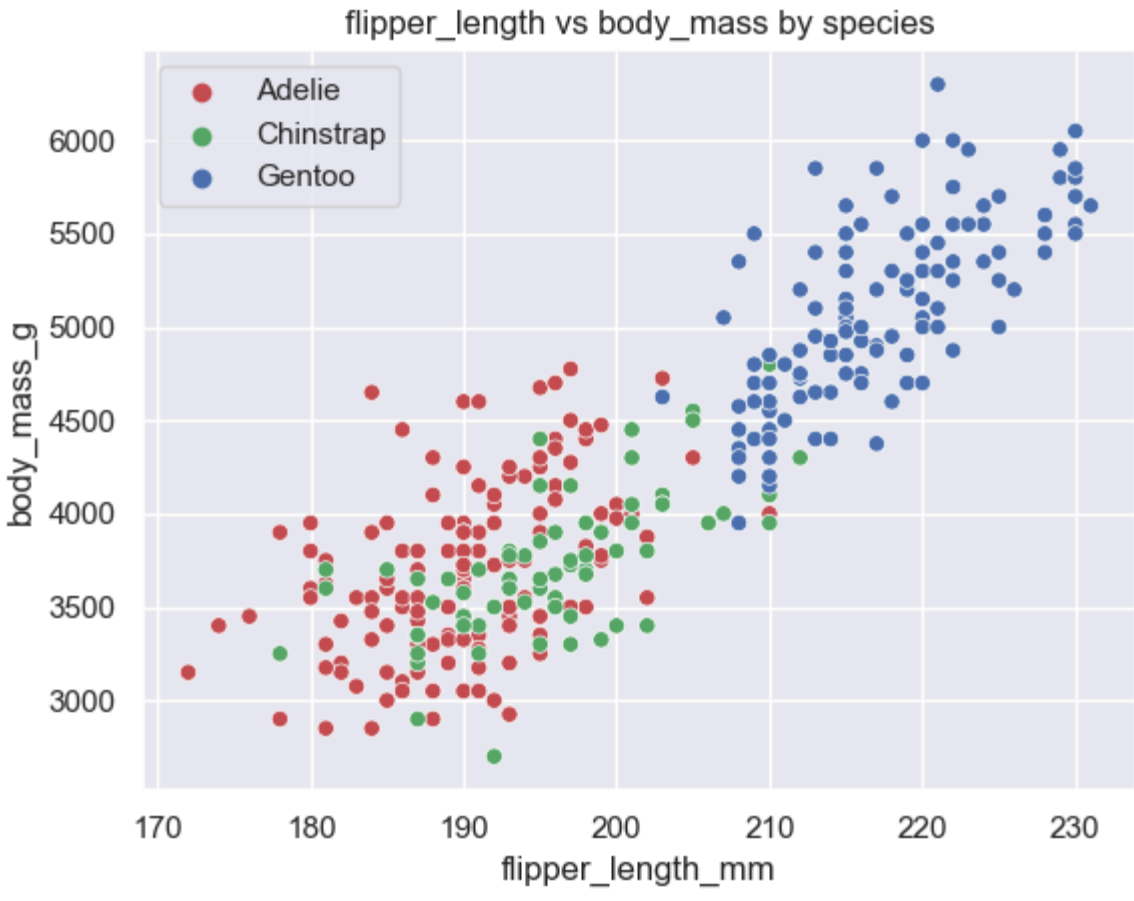


In [38]:
```
clf = KNN(n_neighbors=5)
clf.fit(X,y);
```

In [39]:
```
data['pred_y'] = clf.predict(X)
```

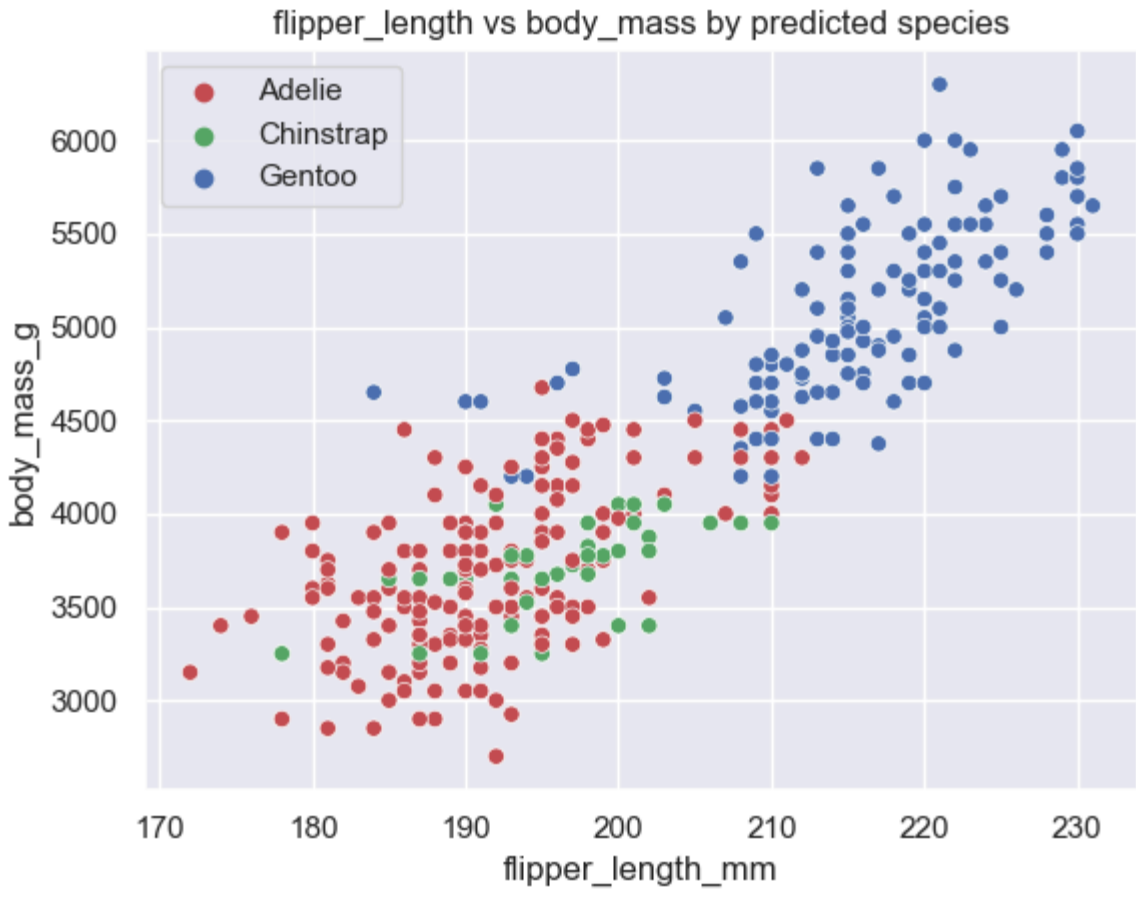
# Original Data Plot

In [8]:
```
sns.scatterplot(data = data, x = 'flipper_length_mm', y='body_mass_g',
plt.title('flipper_length vs body_mass by species')
handles, labels = plt.gca().get_legend_handles_labels()
plt.legend(handles = handles, labels = list(mapping))
plt.show()
```


flipper_length vs body_mass by species
Adelie
Chinstrap
Gentoo
body_mass_g
6000
5500
5000
4500
4000
3500
3000
170
180
190
200
210
220
230
flipper_length_mm

# Predicted Data Plot

In [9]:
```
sns.scatterplot(data = data, x = 'flipper_length_mm', y='body_mass_g',
plt.title('flipper_length vs body_mass by predicted species')
handles, labels = plt.gca().get_legend_handles_labels()
plt.legend(handles = handles, labels = list(mapping))
plt.show()
```

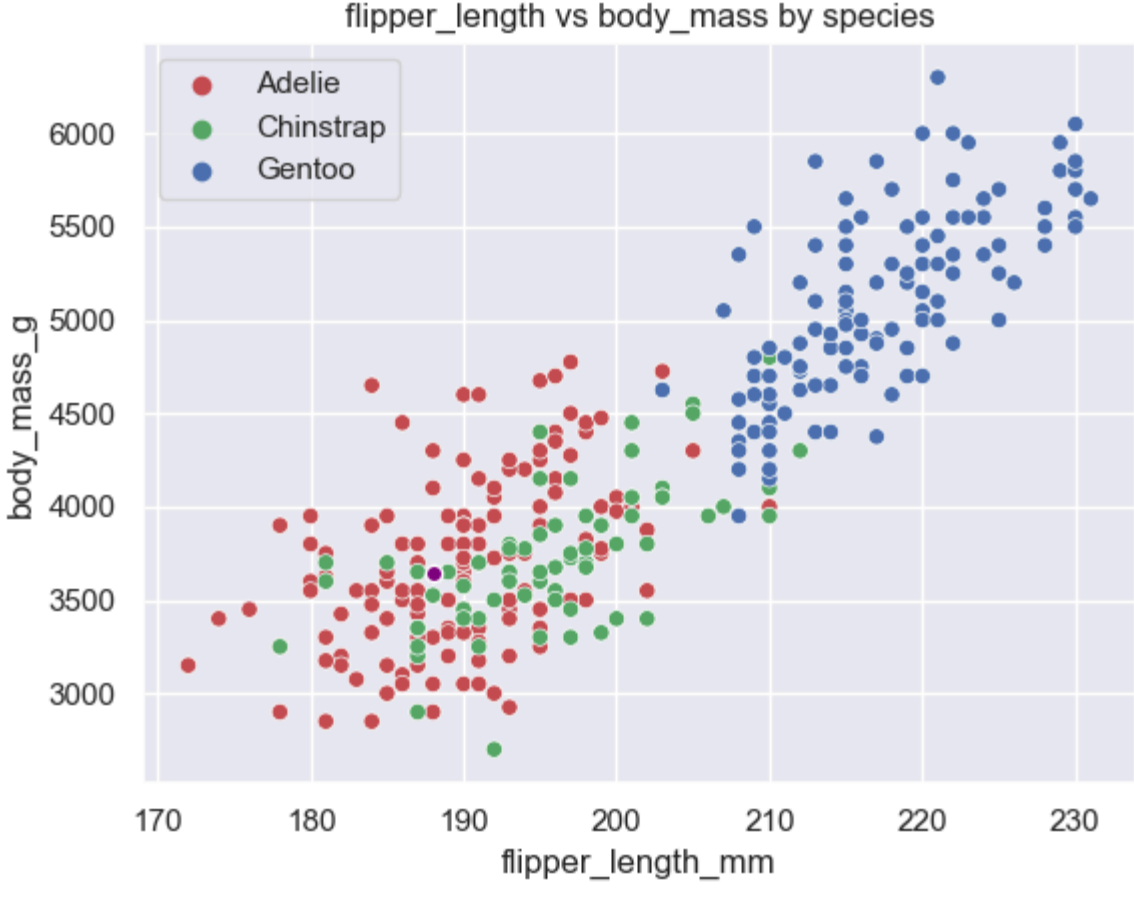

flipper_length vs body_mass by predicted species
Adelie
Chinstrap
Gentoo
body_mass_g
6000
5500
5000
4500
4000
3500
3000
170
180
190
200
210
220
230
flipper_length_mm

# Predicting New Points

What species is most likely for a penguin that has a flipper length of 188 mm and a body mass of 3650 grams?

In [10]:

```
sns.scatterplot(data = data, x = 'flipper_length_mm', y='body_mass_g',
plt.title('flipper_length vs body_mass by species')
handles, labels = plt.gca().get_legend_handles_labels()
plt.legend(handles = handles, labels = list(mapping))
plt.scatter(188, 3650, color='purple', edgecolor='white')
plt.show()
```



# Predicting New Points (cont.)

Our resulting prediction changes depending on the number of neighbors we set for our model.

In [12]:
```
print(f'prediction for 2 neighbors: {output}')
```

```
prediction for 2 neighbors: Chinstrap
```

In [14]:
```
print(f'prediction for 3 neighbors: {output}')
```

```
prediction for 3 neighbors: Chinstrap
```

In [16]:
```
print(f'prediction for 4 neighbors: {output}')
```

```
prediction for 4 neighbors: Adelie
```

In [18]:
```
print(f'prediction for 5 neighbors: {output}')
```

```
prediction for 5 neighbors: Chinstrap
```

# Under the Hood

- When predicting new points, we need to:
    - calculate the distance from every point
    - take the k nearest points into consideration
    - classify according to most common class among neighbors

In [19]:
```
clf = KNN(n_neighbors=4)
clf.fit(X,y)
dists, indices = clf.kneighbors(X = [[3650,188]], n_neighbors=8, returr
```

In [20]:
```
[(reverse_mapping[list(data.iloc[indices[0]]['species'])[x]], dists[0]
```

Out[20]:
```
[('Chinstrap', 1.0),
 ('Chinstrap', 1.0),
 ('Adelie', 2.0),
 ('Adelie', 3.0),
 ('Chinstrap', 5.0),
 ('Chinstrap', 7.0),
 ('Adelie', 25.96150997149434),
 ('Chinstrap', 26.248809496813376)]
```

# Under the Hood (cont.)

When we used a KNN with k=4 and used it to predict the species of a penguin with 188 body_mass and 3650 flipper_length, it followed the process outlined above and resulted in a tie (2 chinstrap, 2 adelie) among its 4 nearest neighbors, which it then arbitrarily broke and classified this point as an Adelie penguin.

Thanks for coming!