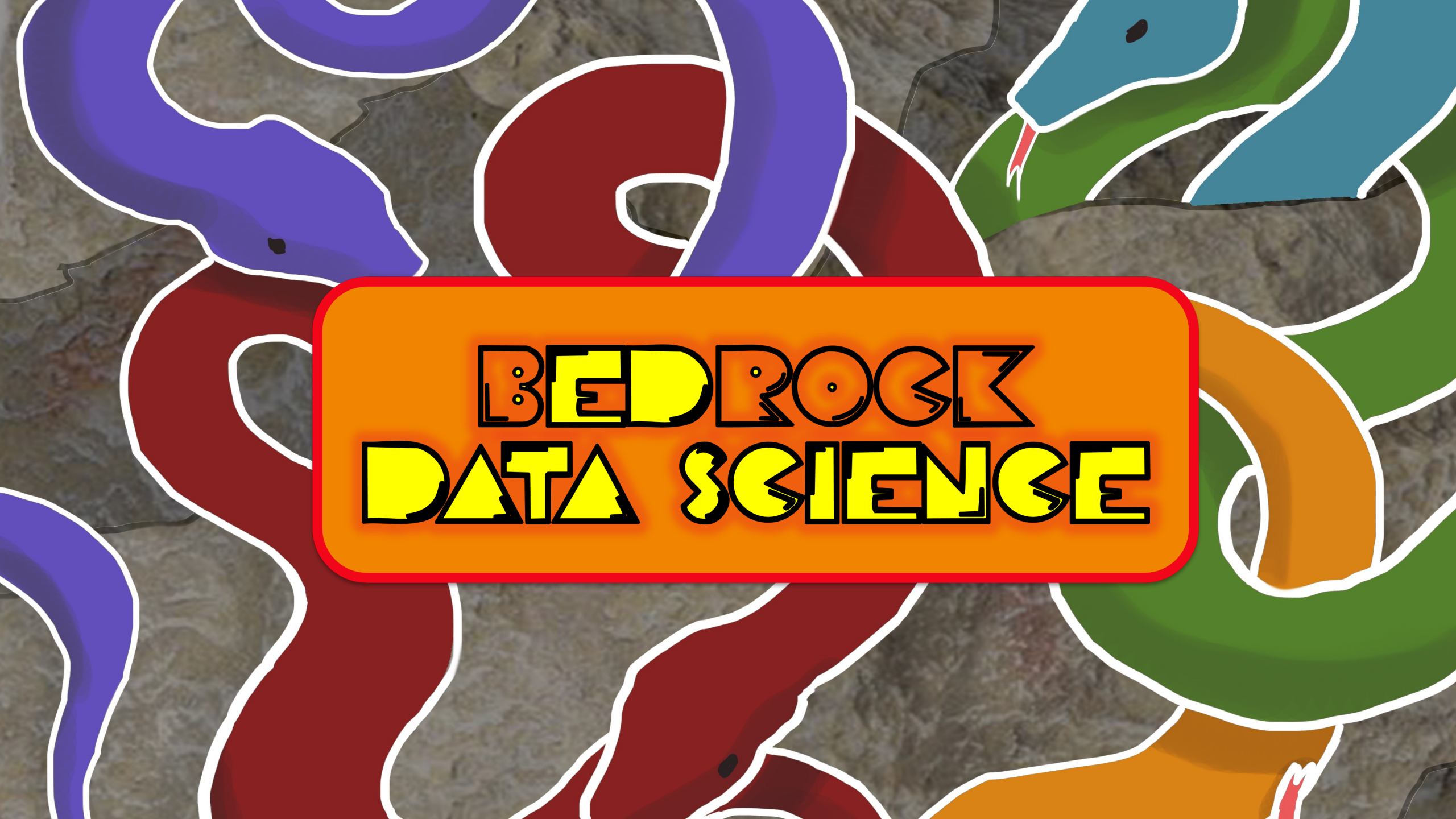

BEDROCK
DATA SCIENCE

BROS

GAME TIME

```
In [48]: footballers = {'ronaldo':7,'messi':10,'gerrard':8}

Out[104]: [('messi', 10), ('gerrard', 8), ('ronaldo', 7)]
```

Options

A. `sorted(footballers.items(),key = lambda x: x[0])`

B. `sorted(footballers.items(),key = lambda x:-x[0])`

C. `sorted(footballers.items(),key = lambda x: x[1])`

D. `sorted(footballers.items(),key = lambda x: -x[-1])`

```
In [48]: footballers = {'ronaldo':7,'messi':10,'gerrard':8}

Out[104]: [('messi', 10), ('gerrard', 8), ('ronaldo', 7)]
```

Options

A. `sorted(footballers.items(),key = lambda x: x[0])`

B. `sorted(footballers.items(),key = lambda x:-x[0])`

C. `sorted(footballers.items(),key = lambda x: x[1])`

D. `sorted(footballers.items(),key = lambda x: -x[-1])`

```
In [47]: def reverse_dictionary(**kwargs):
    ...:     return {v:k for k,v in kwargs.items()}
    ...:
In [48]: footballers = {'ronaldo':7,'messi':10,'gerrard':8}

Out[49]: {7: 'ronaldo', 10: 'messi', 8: 'gerrard'}
```

Options

A. reverse_dictionary(footballers)

B. reverse_dictionary(*footballers)

C. reverse_dictionary(**footballers)

D. reverse_dictionary(***footballers)

```
In [47]: def reverse_dictionary(**kwargs):
    ...:     return {v:k for k,v in kwargs.items()}
    ...:
In [48]: footballers = {'ronaldo':7,'messi':10,'gerrard':8}

Out[49]: {7: 'ronaldo', 10: 'messi', 8: 'gerrard'}
```

Options

A. `reverse_dictionary(footballers)`

B. `reverse_dictionary(*footballers)`

C. `reverse_dictionary(**footballers)`

D. `reverse_dictionary(***footballers)`

```
In [66]: def swiggy_orders(order,foodlist):
    ...:     foodlist.append(order)
    ...:     return foodlist

In [68]: cart  = ['chicken wings’,’glass noodles']
In [69]: new_cart = swiggy_orders('butter chicken',cart)
```

Options

A. cart = ['chicken wings', ‘glass noodles', 'butter chicken']
new_cart = ['chicken wings', 'glass noodles', 'butter chicken']

B. cart = ['chicken wings', 'glass noodles']
new_cart = ['chicken wings', 'glass noodles', 'butter chicken']

C. cart = ['chicken wings', 'glass noodles']
new_cart = ['chicken wings', 'glass noodles']

D. cart = ['chicken wings', 'glass noodles', 'butter chicken']
new_cart = ['chicken wings', 'glass noodles']

```
In [66]: def swiggy_orders(order,foodlist):
    ...:     foodlist.append(order)
    ...:     return foodlist

In [68]: cart  = ['chicken wings’,’glass noodles']
In [69]: new_cart = swiggy_orders('butter chicken',cart)
```

Options

A. cart = ['chicken wings', ‘glass noodles', 'butter chicken']
new_cart = ['chicken wings', 'glass noodles', 'butter chicken']

B. cart = ['chicken wings', 'glass noodles']
new_cart = ['chicken wings', 'glass noodles', 'butter chicken']

C. cart = ['chicken wings', 'glass noodles']
new_cart = ['chicken wings', 'glass noodles']

D. cart = ['chicken wings', 'glass noodles', 'butter chicken']
new_cart = ['chicken wings', 'glass noodles']

Lambda function

```
(SYNTAX) square = lambda x : x**2
```

- Lambda functions (also called **Anonymous functions**) are a concise way to declare functions.
- This pythonic way of defining functions avoids the keyword **RETURN**
- Very helpful when working with mathematical function implementation.

```
In [89]: square = lambda x: x**2


In [90]: square(2)
Out[90]: 4


In [91]: square(2,2)
---------------------------------------------------------------------------
TypeError                                 Trace
back (most recent call last)
<ipython-input-91-fc957e4c04b0> in <module>
----> 1 square(2,2)


TypeError: <lambda>() takes 1 positional
argument but 2 were given
```

Use lambda functions

The list.sort() and sorted() methods can take lambda functions

```
#Example of Sort function(sort in age and in potential)
lst = [('De Ligt',19,92.1), ('Alexander-Arnold',20,90.5),
       ('Sancho',19,94.0),('Havertz',20,93.2),('Rodrygo',18,91.4)]

lst.sort(key=lambda x:x[1]) #Sort by age
print(lst)
lst.sort(key=lambda x:x[2]) #Sort by potential
print(lst)
>>>
[('Rodrygo', 18, 91.4), ('De Ligt', 19, 92.1), ('Sancho', 19, 94.0), ('Alexander-Arnold',
20, 90.5), ('Havertz', 20, 93.2)]

[('Alexander-Arnold', 20, 90.5), ('Rodrygo', 18, 91.4), ('De Ligt', 19, 92.1), ('Havertz',
20, 93.2), ('Sancho', 19, 94.0)]
```

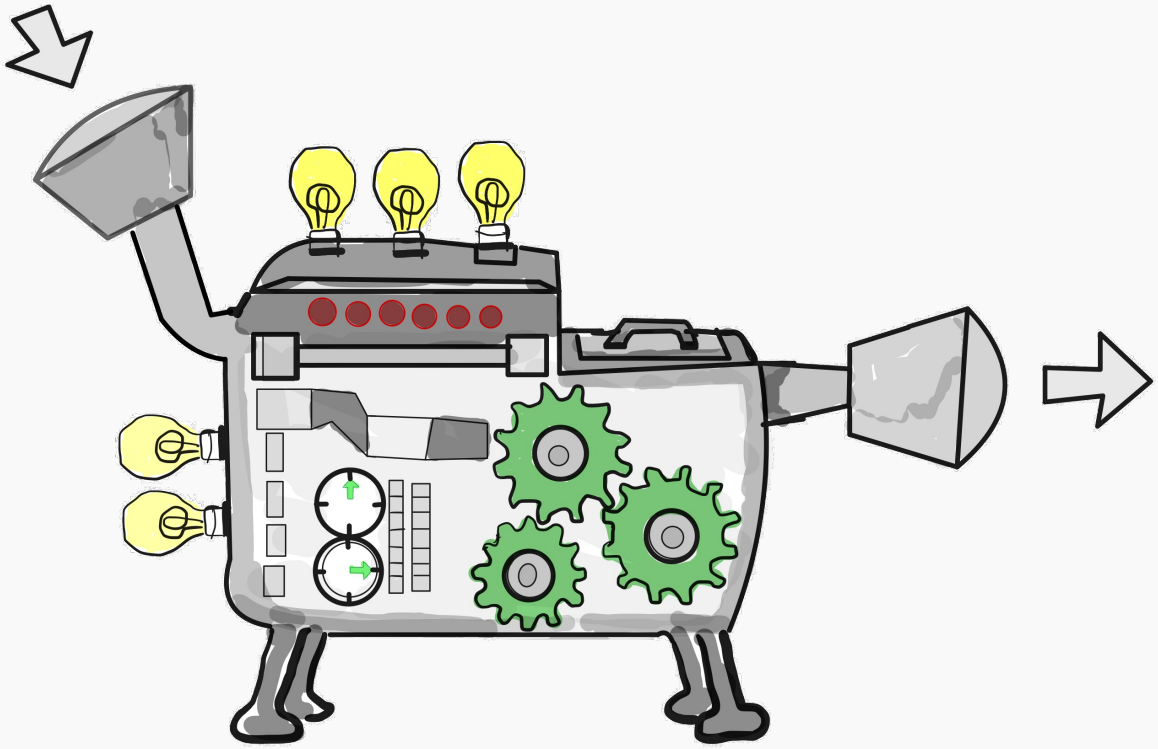
It is NOT compulsory to use a lambda function. Any regular function would do as well

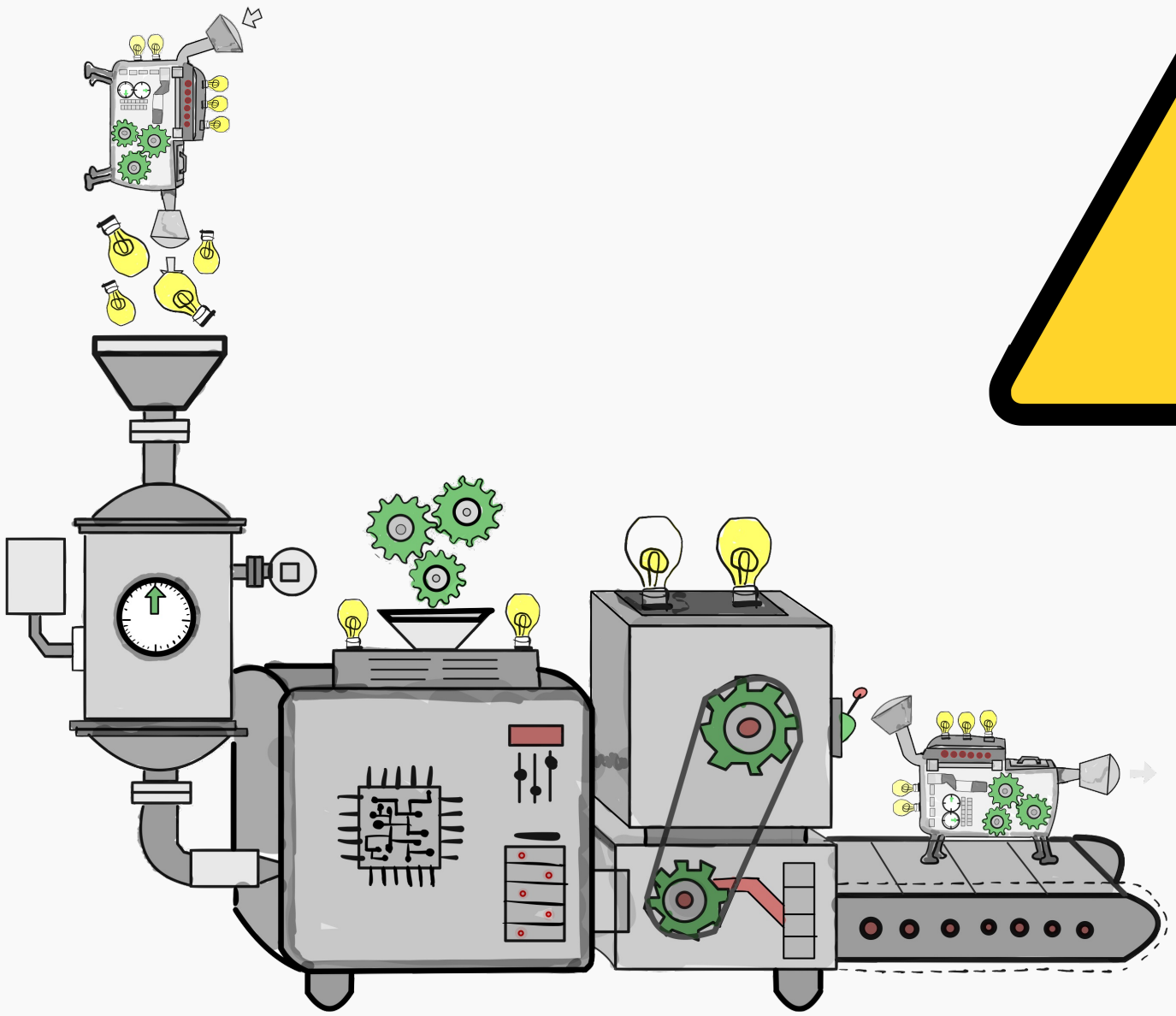
Use lambda functions

- Certain methods/functions exist which make it very easy to apply lambda functions on elements of iterable objects.
- map(), filter() both take a function as an input. (see e.g. below)

```
## Usage of lambda functions

#Example of Map function(calculate discount)
a = list(map(lambda x: x - 0.2*x, [120,300,150,3000,42.55,67.11]))

#Example of Filter function(get all 'back' positions)
b = list(filter(lambda x: 'B' in x, ['LB', 'CB', 'ST','CAM','CMD','GK','RB']))

print(a)
print(b)
>>>
[96.0, 240.0, 120.0, 2400.0, 34.04, 53.688]
['LB', 'CB', 'RB']
```

Advanced Functions

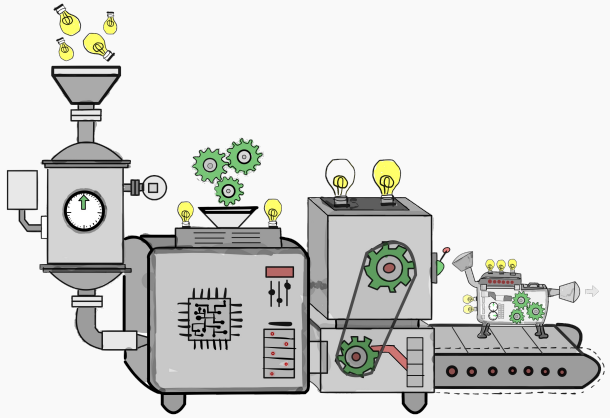




Advanced functions

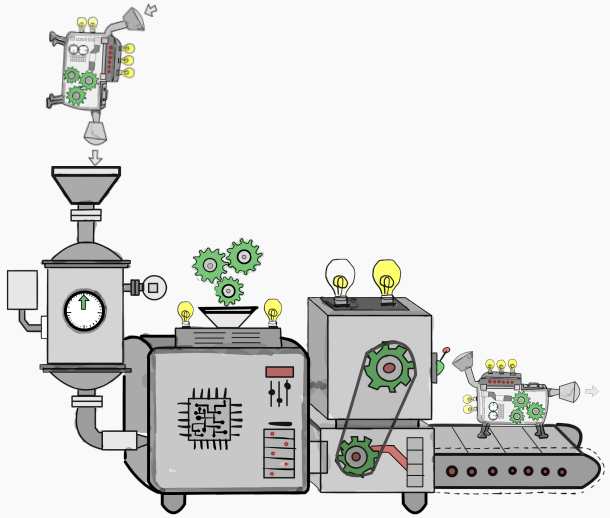
Level #1

Number in, Function out



Level #2

Function in, Function out



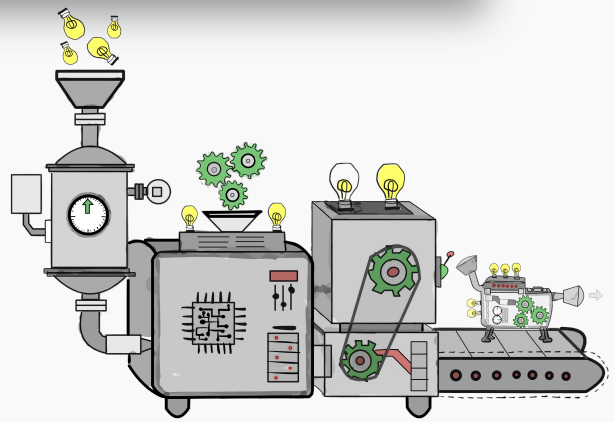
Level #1 - Number in, function out

- Combine a lambda function and use it inside a function to declare an even higher level of generalization.
- It means you are now returning a function, the specifics of which are decided by the input to higher function.

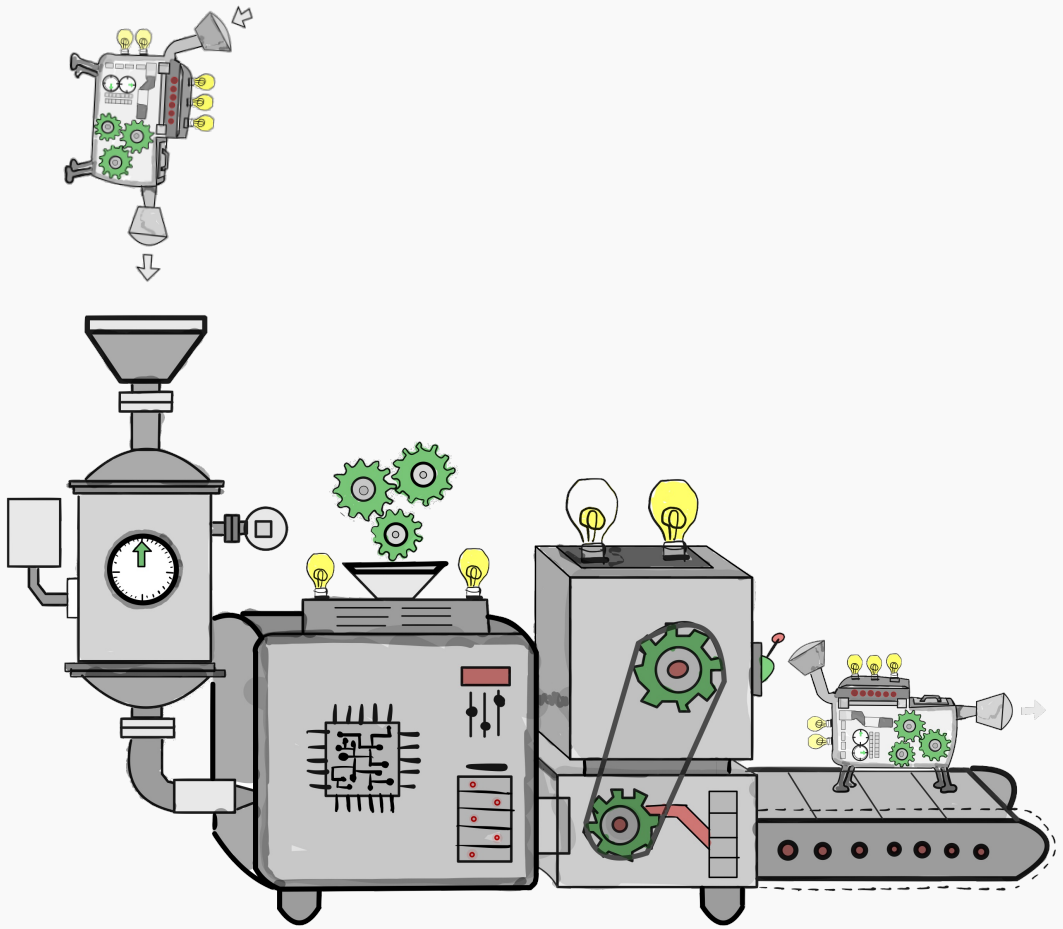
You are returning a function here

```
def multiplier(n):
    func = lambda x:x*n
    return func

doubler = multiplier(2)
tripler = multiplier(3)

print(doubler(100))
print(tripler(100))
>>>
200
300
```

It is an n-multiplier function. Based on input n the output function will vary



Level #2: Function in, Function out



Level #2 – Function in, Function out

```
In [19]: def decorator_function(original_function):
    ...:     def packaging(x):
    ...:         print(”Tao's function. All rights reserved")
    ...:         return original_function(x)
    ...:     return packaging
```

Taking a function as input

You are returning a function here

Level #2 – Function in, Function out

```
In [19]: def decorator_function(original_function):
    ...:     def packaging(x):
    ...:         print("Tao's function. All rights reserved")
    ...:         return original_function(x)
    ...:     return packaging

In [21]: def square(x):
    ...:     return x**2

In [27]: square(5)
Out[27]: 25
```

unprocessed
function call

Level #2 – Function in, Function out

```
In [19]: def decorator_function(original_function):
    ...:     def packaging(x):
    ...:         print("Tao's function. All rights reserved")
    ...:         return original_function(x)
    ...:     return packaging

In [21]: def square(x):
    ...:     return x**2

In [27]: square(5)
Out[27]: 25

In [22]: processed_square = decorator_function(square)
```

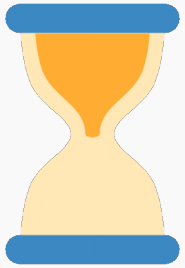
Remember!
decorator_function()
returns a function

after processing using the decorator function

Level #2 – Function in, Function out

```
In [19]: def decorator_function(original_function):
    ...:     def packaging(x):
    ...:         print("Tao's function. All rights reserved")
    ...:         return original_function(x)
    ...:     return packaging

In [21]: def square(x):
    ...:     return x**2

In [27]: square(5)
Out[27]: 25


In [22]: processed_square = decorator_function(square)

In [23]: processed_square(5)
Tao's function. All rights reserved
Out[23]: 25
```

Function call includes the copyright message



Digestion Time

Function Decorators

```
In [19]: def decorator_function(original_function):
    ...:     def packaging(x):
    ...:         print("Tao's function. All rights reserved")
    ...:         return original_function(x)
    ...:     return packaging
```

Without decorator

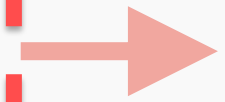
```
square = lambda x:x**2
```

```
square_sum = decorator_function(square)
```

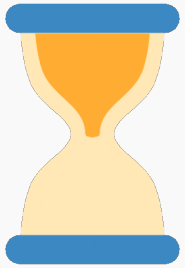
The function you declared below the decorator gets written over. Now it no longer takes only one input

Using decorator

```
@decorator_function
def square(x):
    return x**2
```



```
square = decorator_function(square)
```



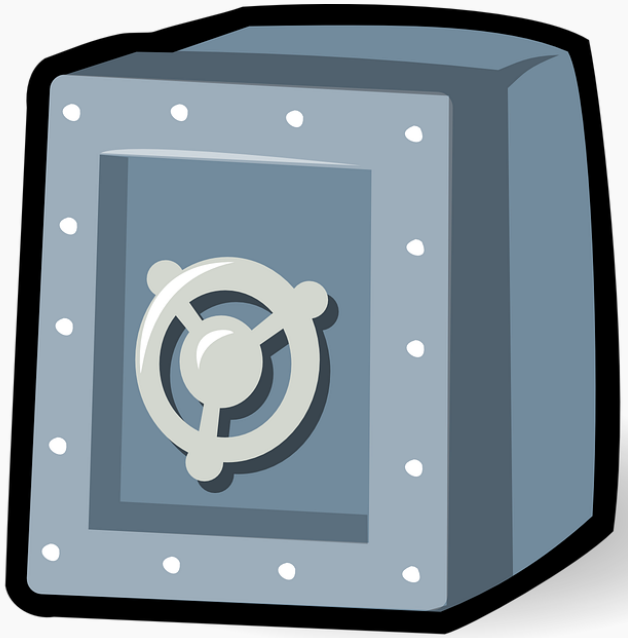
Digestion Time

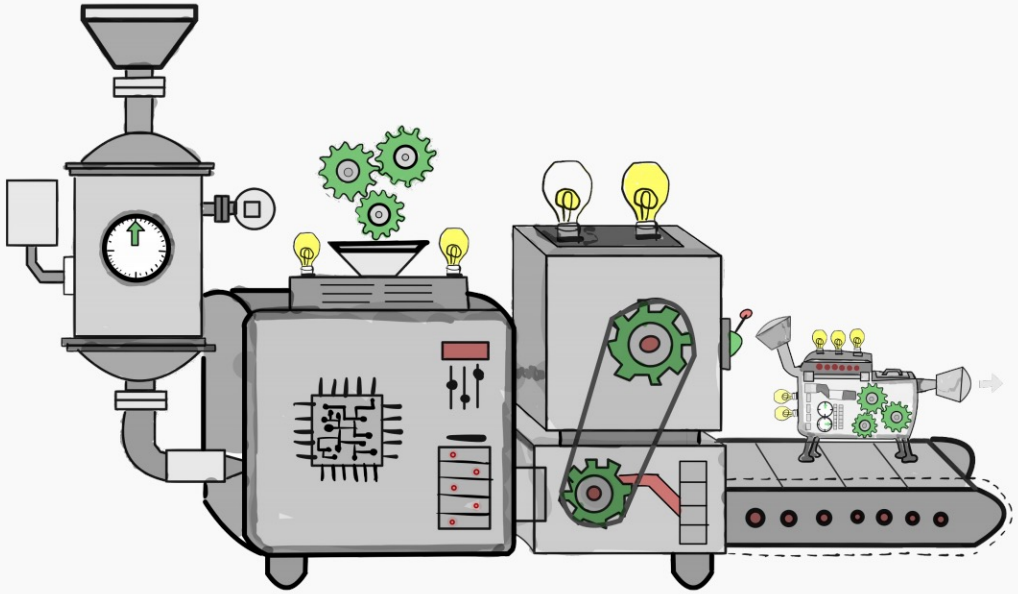
Session 4: Python Classes

Session 3

```
# Variable assignments
>>> x = 5
>>> y = x **(2)
>>> z = x+y
```

```
# functions
>>> def myfunc():
....
....
    return ...
```





Demo

What will happen if I run the following?

```
Hi = 5.2324
Hi.__floordiv__(2)
```

Everything in python is a class object instance

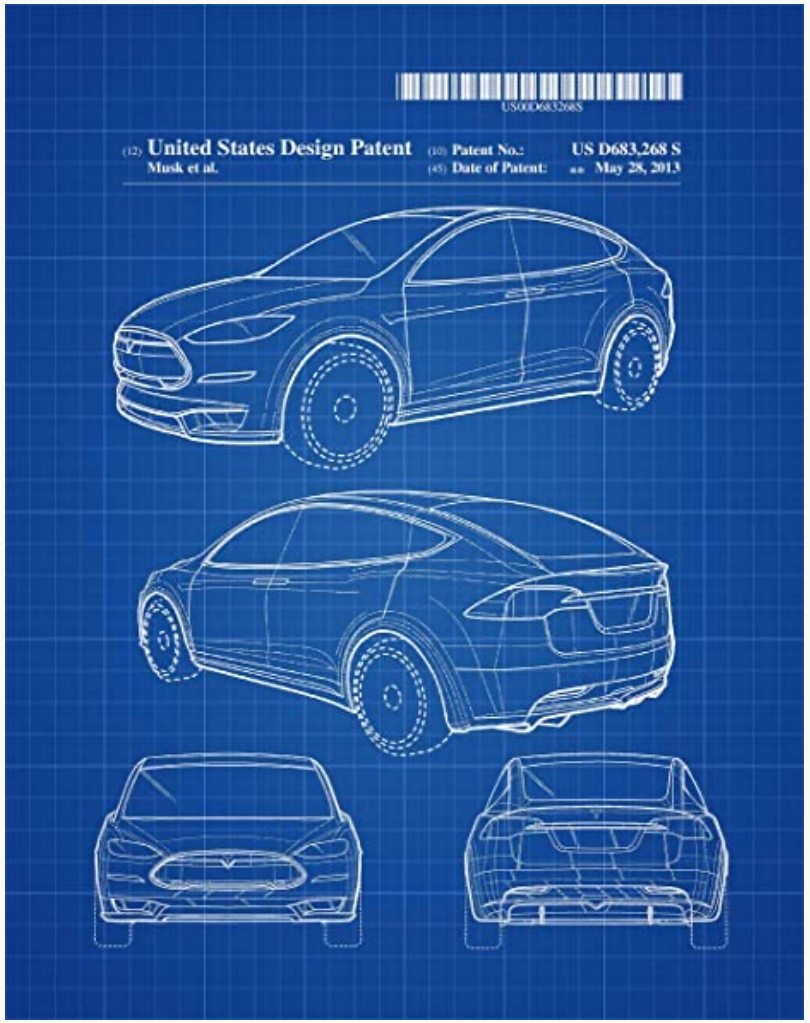
```
# Variables
>>> x = 5
>>> y = x **(2)
```

```
# functions
>>> def myfunc():
....
....
    return ...
```

```
# classes
In [18]: class myclass():
...:
...:     def
__init__(self):
...:     self.x = 5
...:     ...
```

Lets start with traditional analogies.

The class template is often called a "blueprint’’

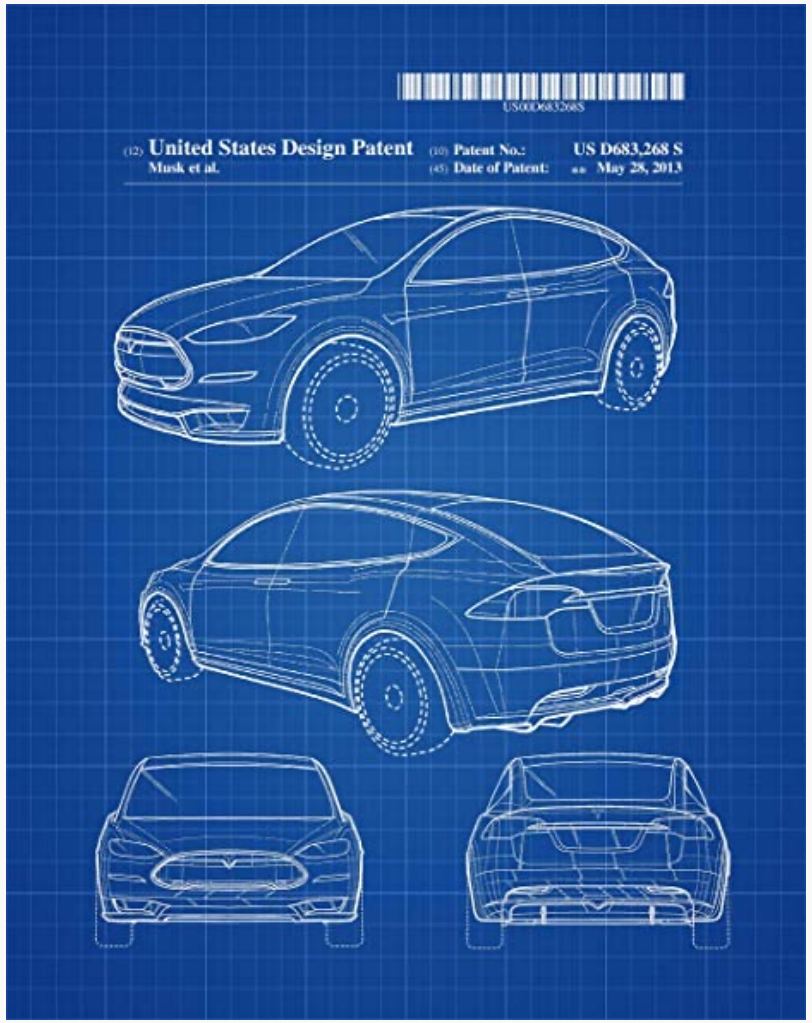


Lets start with traditional analogies.

The class template is often called a "blueprint" or a ***recipe***

Lets start with traditional analogies.

```
# Create a class template
class Model_S:
    year = 2013
    <statement 2>
    […]
    <statement n>
```



Lets start with traditional analogies.

We instantiate the class object instance with function notation

```
# Create a class template
class Model_S:
    Year = 2013
    <statement 2>
    […]
    <statement n>

# Create a class instance
my_tesla = Model_S()
```

Classes vs Instances

```
class Cookie:
    """A generic cookie class'
    cal_per_gram  = {"sugar"
                     "co
                     "choc

    def __init__(self, cookie_
        self.cookie_type,  sel

    @staticmethod
    def calc_cal(ingredient_di
        total_calories = 0
        for ingredient, amount
            total_calories +=
        return total_calories

class Chocolate_chip(Cookie):
    """A chocolate chip cookie
    def __init__(self, chocola
        self.cookie_type = "ch
        self.num_choc_chips =

        #ingredients are in gr
        self.ingredients = {

        self.num_cookies = 16

    def __str__(self):
        total_calories = self.
        cal_per_cookie = round
        cchips = self.ingredie
        return 'I am a {} with
                self.cookie_ty
```


```
plain_cookie = Chocolate_chip(chocolate_chips = 0)
normal_cookie = Chocolate_chip(chocolate_chips = 5)
my_cookie = Chocolate_chip(chocolate_chips = 10)
```

Source: https://sallysbakingaddiction.com/chewy-chocolate-chip-cookies/

Object Oriented Programming (OOP) languages

What is object oriented programming?

OOP "is a coding paradigm based on the concept of objects, which can contain data and code."

Image Source: https://en.wikipedia.org/wiki/Object-oriented_programming

Object Oriented Programming (OOP) languages

What is object oriented programming?

OOP "is a coding paradigm based on the concept of objects, which can contain data and code."

Examples of OOP languages:

- Python, C++, R, Java, C#

Image Source: https://en.wikipedia.org/wiki/Object-oriented_programming

Object Oriented Programming (OOP) languages

What is object oriented programming?

OOP "is a coding paradigm based on the concept of objects, which can contain data and code."

Examples of OOP languages:

- Python, C++, R, Java, C#

Examples of non-OOP languages:

- Assembler, C, Fortran

Image Source: https://en.wikipedia.org/wiki/Object-oriented_programming

Object Oriented Programming (OOP) languages

What is object oriented programming?

OOP "is a coding paradigm based on the concept of objects, which can contain data and code."

Examples of OOP languages:

- Python, C++, R, Java, C#

Examples of non-OOP languages:

- Assembler, C, Fortran

data fields == *attributes*

functions == *methods*

Image Source: https://en.wikipedia.org/wiki/Object-oriented_programming

Outline for Section 4: Python Classes and OOP

1. The anatomy of a class
    1. Classes Syntax
    2. What is '`self`' ?
    3. Methods & Attributes
2. Why Classes ?
    1. A proof by contradiction
    2. Looking at `sklearn`
3. Dunder methods:
        1. `__init__`, `__call__`, `__str__`, `__repr__`
        2. Iterable dunder methods
4. Bonus material

Class Basics

- We can define an object instance like so

```
# Create a class template
class Cookie:
    # a class method
    sound = "crunch"
    def make_sound(self):
        print(self.sound, !)
```

```
# Create a class instance
>>> my_cookie = Cookie()
```

Class Basics

- We can access attributes [instance].[attribute]

```
# Create a class template
class Cookie:
    # a class method
    sound = "crunch"
    def make_sound(self):
        print(self.sound, !)
```

```
# Create a class instance
>>> my_cookie = Cookie()

# explore a class
attribute
>>> my_cookie.sound
"crunch”
```

Class Basics

- we can call object methods with function notation

```
# Create a class template
class Cookie:
    # a class method
    sound = "crunch"
    def make_sound(self):
        print(self.sound, !)
```

```
# Create a class instance
>>> my_cookie = Cookie()

# explore a class
attribute
>>> my_cookie.sound
"crunch"

# call an instance method
>>> my_cookie.make_sound()
"crunch!"
```

Instance Methods

- Remember methods are functions

```
# Create a class template
class Model_S:
    def make(self):
        […]
```

Instance Methods

- Remember methods are functions
- We can call them just like functions!

```
# Create a class template
class Model_S:
    def make(self):
        print(f"This car is a {self.year} Tesla Model S")

# Create a class instance
>>>  my_tesla = Model_S()
>>>  my_tesla.make()
"This car is a 2013 Tesla Model S"
```

Instance Methods

What is self?

- The instance passes itself to its instance methods upon being called.

```
# Create a class template
class Model_S:
    year = 2013
    def make(self):
        print(f"This car is a {self.year} Tesla Model S")

# Create a class instance
>>>  my_tesla = Model_S()
>>>  my_tesla.make(2013)
"This car is a 2013 Tesla Model S"
```

Instance Methods

What is self?

- The instance passes itself to its instance methods upon being called.

An F string is a convenient way to format strings with variables, more on this in section 5

```
# Create a class template
class Model_S:
    def make(self, year):
        print(f"This car is a {year} Tesla Model S")

# Create a class instance
>>>  my_tesla = Model_S()
>>>  my_tesla.make(2013)
"This car is a 2013 Tesla Model S"
```

Object self reference

What is self?

- The instance passes itself to its instance methods upon being called.
- with self we can assign instance attributes

```
# Create a class template
class Model_S:
    def make(self, year):
        self.model = str(year) + “model S with autopilot”
        […]
```

Object self reference

What is self?

- The instance passes itself to its instance methods upon being called.
- with self we can assign instance attributes
- or call instance methods

```
# Create a class template
class Model_S:
    year = 2013
    def make(self):
        self.make = “model S with autopilot”
        self.vroom()
    def vroom(self):
        print(“VROOM”)
```

Instance method

```
In [36]: class MyClass:
    ...:
    ...:     def myfunc(self):
    ...:         print('Object method')
    ...:

In [37]: obj = MyClass()

In [38]: obj.myfunc()
Object method
```

```
obj.myfunc()
```

Instance method

```
In [36]: class MyClass:
    ...:
    ...:     def myfunc(self):
    ...:         print('Object method')
    ...:
```

```
In [37]: obj = MyClass()
```

```
In [38]: obj.myfunc()
Object method
```

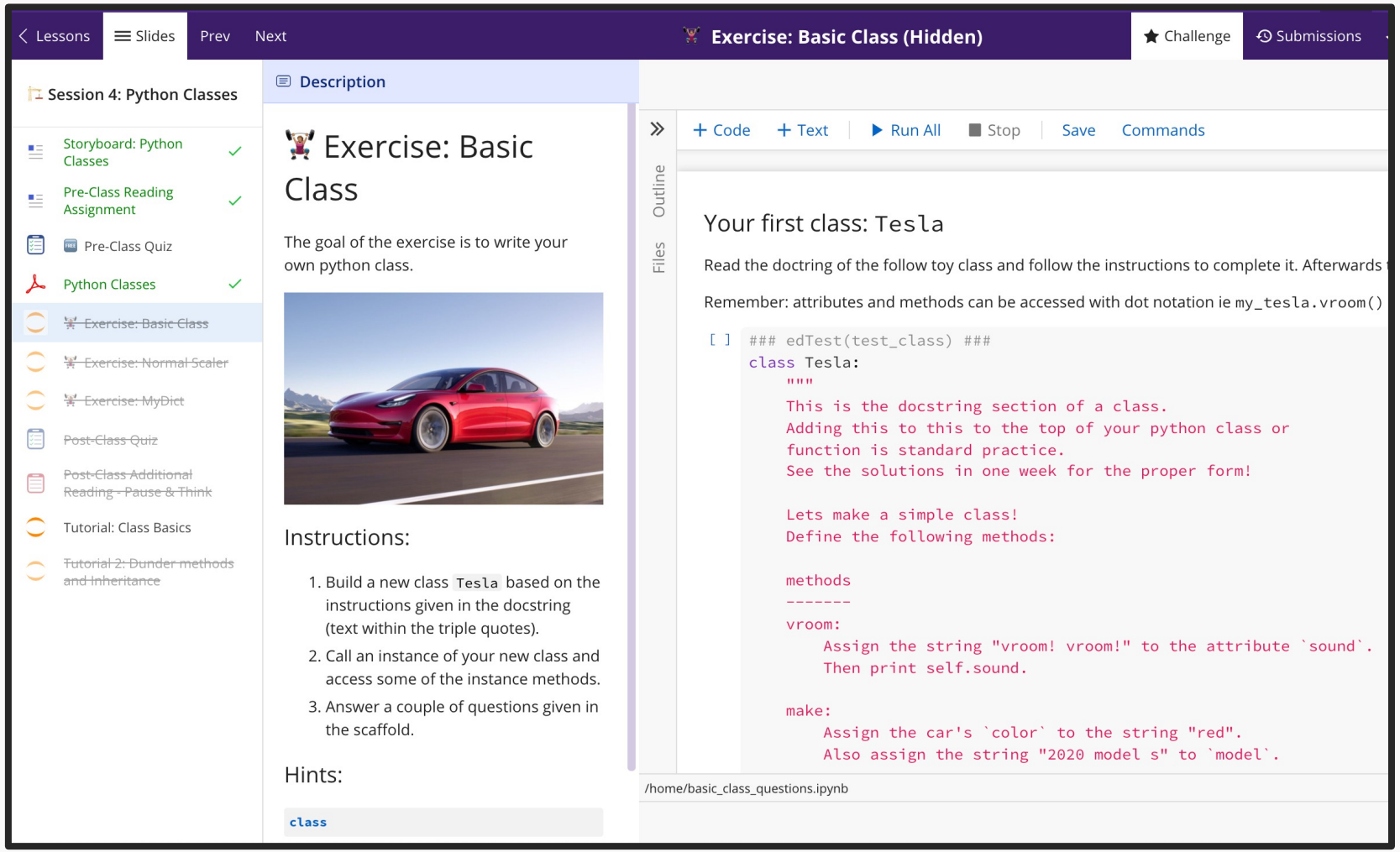
```
In [39]: MyClass.myfunc(obj)
Object method
```

```
obj.myfunc()
```

```
MyClass.myfunc(obj)
```

Exercise 1

Build your own Tesla Class!



However these analogies are problematic because they suggest that classes are immutable. ***They are not.*** ***Instances and classes are mutable.***

Modifying Attributes

```
# object attributes are easily overwritten.
>>>  my_cookie.sound
"crunch"

>>>  my_cookie.sound = "pop"
>>>  my_cookie.sound
"pop”
```

Modifying Attributes

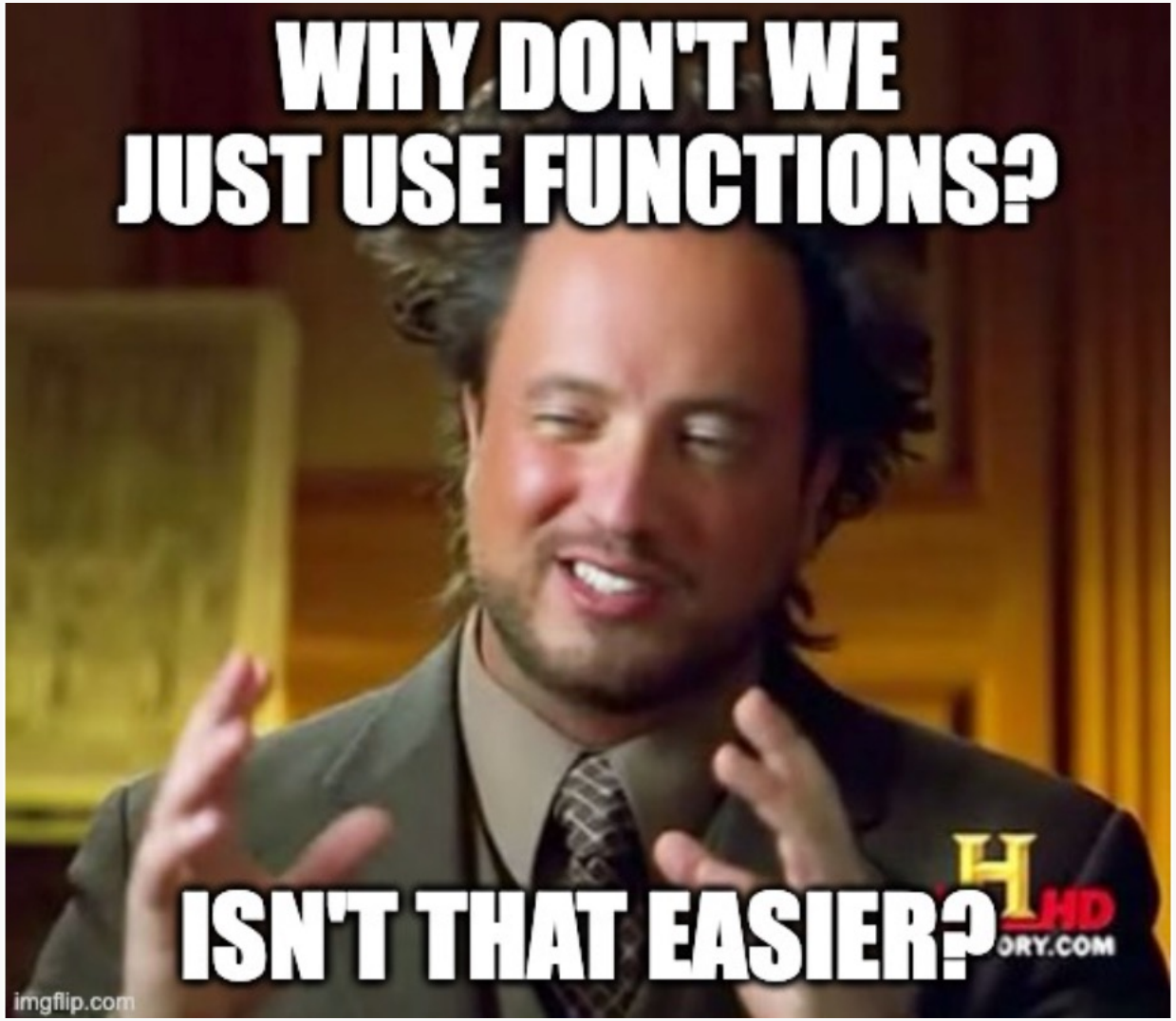
```
# object attributes are easily overwritten.
>>>  my_cookie.sound
"crunch"

>>>  my_cookie.sound = "pop"
>>>  my_cookie.sound
"pop"

# We can also easily add methods.
>>>  def get_calories(self):
        print(64)
>>>  my_cookie.get_cal = get_calories
>>>  my_cookie.get_cal()
"64"
```

Act II Abstraction

Definition 1:

The main purpose of **abstraction** is hiding the unnecessary details from the users. ... It is one of the most important concepts of OOPs.

The problem



Lets do our own ***Proof by contradiction***

Let’s try a ***proof by contradiction.*** First assume something is false, then find logical inconsistency.

PROOF

BY

CONTRADICTION

Application: Data Standardization

Standardizing data is important for two reasons:

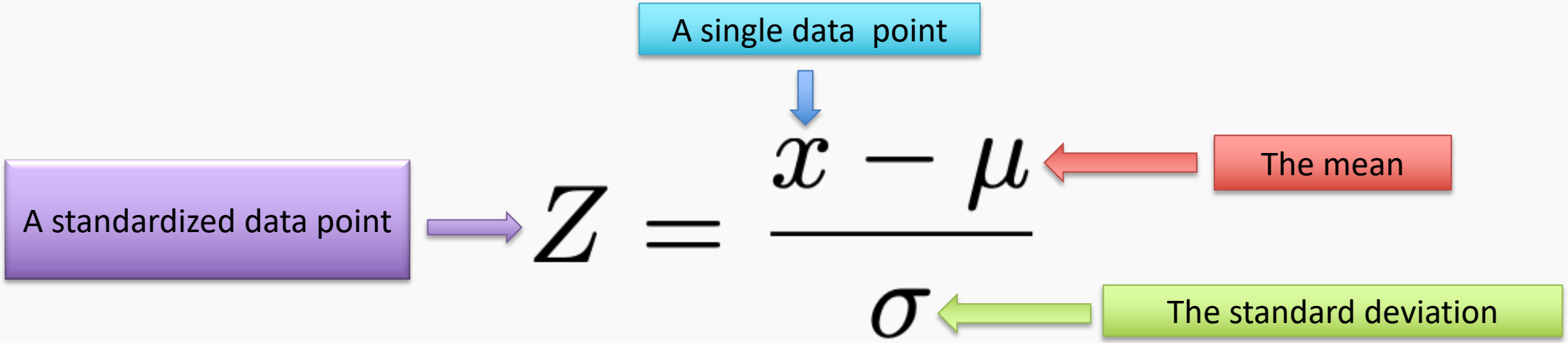
1. Bringing data to the same scale
2. Machine learning Models like NNs generally work better with small numbers because of ***computer precision***

$$Z = \frac{x - \mu}{\sigma}$$

Application: Data Normalization

Standardizing data is important for two reasons:

1. Bringing data to the same scale
2. Machine learning Models like NNs generally work better with small numbers because of ***computer precision***



Without functions:

```
# calculate standardized dataset 1
>>>  data1 = [3.5, 4.2, 1.2, 7.5, 2.2]

# calculate the dataset mean and standard deviation
```

Without functions:

```
# calculate standardized dataset 1
>>>  data1 = [3.5, 4.2, 1.2, 7.5, 2.2]

# calculate the dataset mean and standard deviation
>>> n = len(data)
>>> mean1 = sum(data)/n
>>> var1 = sum([(point - mean1)**2/(n - 1)) for point in data1])
>>> std_dev1 = math.sqrt(var1)
```

Without functions:

```
# calculate standardized dataset 1
>>>  data1 = [3.5, 4.2, 1.2, 7.5, 2.2]

# calculate the dataset mean and standard deviation
>>> n = len(data)
>>> mean1 = sum(data)/n
>>> var1 = sum([(point – mean1)**2/(n - 1)) for point in data1])
>>> std_dev1 = math.sqrt(var1)

# calculated the standardized data
>>> norm_data = [(dp - mean1)/std_dev1 for dp in data1]
```

Without functions:

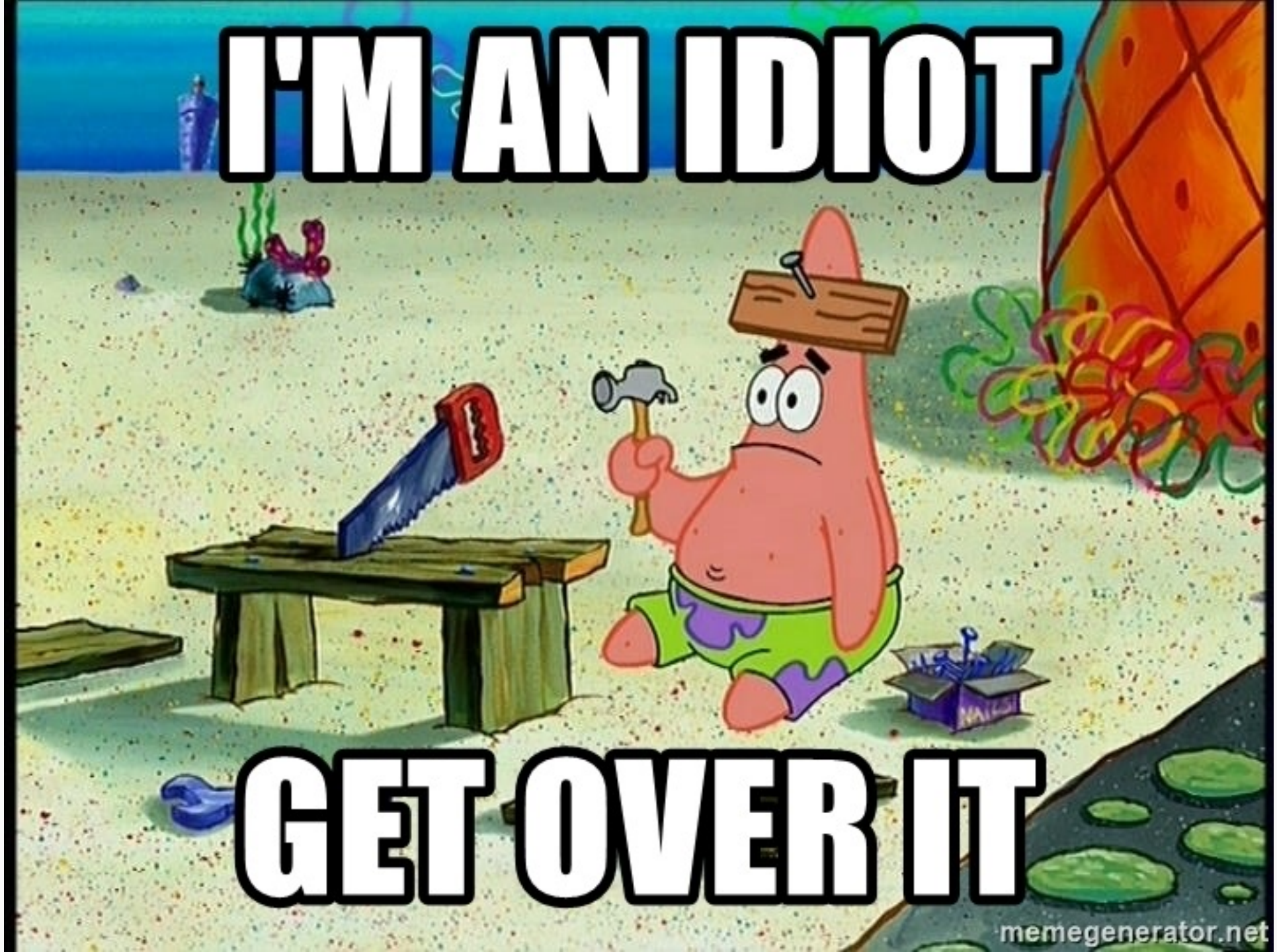
```
# calculate standardized dataset 1
>>>  data1 = [3.5, 4.2, 1.2, 7.5, 2.2]

# calculate the dataset mean and standard deviation
>>> n = len(data)
>>> mean1 = sum(data)/n
>>> var1 = sum([(point – mean1)**2/(n - 1)) for point in data1])
>>> std_dev1 = math.sqrt(var1)

# calculated the standardized data
>>> norm_data = [(dp - mean1)/std_dev1 for dp in data1]

# calculate standardized dataset 2
>>> data = [4.2, 0.1, 8.4, 7.5, 2.9]
>>> mean2 = sum(data)/len(data)
>>> var2 = sum([(point – mean2)**2 for point in data])
>>> std_dev2 = math.sqrt(data_variance)
>>> norm_data2 = [(dp - mean2)/std_dev2 for dp in std_dev2]

#we have to write the same five lines over and over again for every
new dataset, how annoying!
```

Without functions:

```
# calculate standardized dataset 1
>>>  data1 = [3.5, 4.2, 1.2, 7.5, 2.2]

# calculate the dataset mean and standard deviation
>>> n = len(data)
>>> mean1 = sum(data)/n
>>> var1 = sum([(point - mean1)**2/(n - 1)]) for point in data1])
>>> std_dev1 = math.sqrt(var1)

# calculated the standardized data
>>> norm_data = [(dp - mean1)/std_dev1 for dp in data1]

# calculate standardized dataset 2
>>> data = [4.2, 0.1, 8.4, 7.5, 2.9]
>>> mean2 = sum(data)/len(data)
>>> var2 = sum([(point - mean2)**2 for point in data])
>>> std_dev2 = math.sqrt(data_variance)
>>> norm_data2 = [(dp - mean2)/std_dev2 for dp in std_dev2]

#we have to write the same five lines over and over again for every
new dataset, how annoying!
```

Patrick says:

"Functions bad!"


I'M AN IDIOT
GET OVER IT
memegenerator.net

Standardization II: With functions

```
# functions (psuedo-code)
>>>  def mean(data):
         pass
>>>  def std(data):
         pass
>>>  def standardize(data):
         pass
```

Standardization II: With functions

```
# functions (psuedo-code)
>>>  def mean(data):
         return sum(data) / len(data)
>>>  def std(data):
         pass
>>>  def standardize(data):
         pass
```

Standardization II: With functions

```
# functions (psuedo-code)
>>>  def mean(data):
         return sum(data) / len(data)
>>>  def std(data):
         var = sum([(dp - mean(data))**2/(len(data) - 1) for dp in data])
         return math.sqrt(var)
>>>  def standardize(data):
         pass
```

Standardization II: With functions

```
# functions (psuedo-code)
>>>  def mean(data):
         return sum(data) / len(data)
>>>  def std(data):
         var = sum([(dp - mean(data))**2/(len(data) - 1) for dp in data])
         return math.sqrt(var)
>>>  def standardize(data):
         return [(dp - mean(data))/std(data) for dp in data]
```

Standardization II: With functions

```
# functions (psuedo-code)
>>>  def mean(data):
         return sum(data) / len(data)
>>>  def std(data):
         var = sum([(dp – mean(data))**2/(len(data) - 1) for dp in data])
         return math.sqrt(var)
>>>  def standardize(data):
         return [(dp – mean(data))/std(data) for dp in data]

>>> data = [3.5, 4.2, 1.2, 7.5, 2.2]
```

Standardization II: With functions

```
# functions (psuedo-code)
>>>  def mean(data):
         return sum(data) / len(data)
>>>  def std(data):
         var = sum([(dp – mean(data))**2/(len(data) - 1) for dp in data])
         return math.sqrt(var)
>>>  def standardize(data):
         return [(dp – mean(data))/std(data) for dp in data]

>>> data = [3.5, 4.2, 1.2, 7.5, 2.2]
>>> norm_data = standardize(data)
```

Standardization II: With functions

```
# functions (psuedo-code)
>>>  def mean(data):
         return sum(data) / len(data)
>>>  def std(data):
         var = sum([(dp – mean(data))**2/(len(data) - 1) for dp in data])
         return math.sqrt(var)
>>>  def standardize(data):
         return [(dp – mean(data))/std(data) for dp in data]

>>> data = [3.5, 4.2, 1.2, 7.5, 2.2]
>>> norm_data = standardize(data)
>>> norm_data_mod = f(norm_data)
```

Application: Data Normalization

Standardizing data is reversible.

Standardize a data point

$$Z = \frac{x - \mu}{\sigma}$$

Bring it back to the original scale

$$x = \sigma z + \mu$$

μ : the mean
σ : the standard deviation
Z : a standardized data point
x : a data point

Standardization II: With functions

```
# functions (psuedo-code)
>>>  def mean(data):
         return sum(data) / len(data)
>>>  def std(data):
         var = sum([(dp – mean(data))**2/(len(data) - 1) for dp in data])
         return math.sqrt(var)
>>>  def standardize(data):
         return [(dp – mean(data))/std(data) for dp in data]

>>> data = [3.5, 4.2, 1.2, 7.5, 2.2]
>>> norm_data = standardize(data)
>>> norm_data_mod = f(norm_data)

# But what if we want to go back to the original data?!
>>> mean_ = mean(data)
>>> std_  = std(data)
```

Standardization II: With functions

```
# functions (psuedo-code)
>>>  def mean(data):
         return sum(data) / len(data)
>>>  def std(data):
         var = sum([(dp – mean(data))**2/(len(data) - 1) for dp in data])
         return math.sqrt(var)
>>>  def standardize(data):
         return [(dp – mean(data))/std(data) for dp in data]

>>> data = [3.5, 4.2, 1.2, 7.5, 2.2]
>>> norm_data = standardize(data)
>>> norm_data_mod = f(norm_data)

# But what if we want to go back to the original data?!
>>> mean_ = mean(data)
>>> std_  = std(data)
>>> modified_data = [data_point*std_ + mean_ for data_point in norm_data_mod ]
```

Standardization II: With functions

```
# functions (psuedo-code)
>>>  def mean(data):
         return sum(data) / len(data)
>>>  def std(data):
         var = sum([(dp - mean(data))**2/(len(data) - 1) for dp in data])
         return math.sqrt(var)
>>>  def standardize(data):
         return [(dp - mean(data))/std(data) for dp in data]
>>>  def unstd(data):
         pass
```

Standardization II: With functions

```
# functions (psuedo-code)
>>>  def mean(data):
         return sum(data) / len(data)
>>>  def std(data):
         var = sum([(dp - mean(data))**2/(len(data) - 1) for dp in data])
         return math.sqrt(var)
>>>  def standardize(data):
         return [(dp - mean(data))/std(data) for dp in data]
>>>  def unstd(data, data_mean, data_std):
         return [(dp * data_std) + data_mean for dp in data]
```

Standardization II: With functions

```
# functions (psuedo-code)
>>>  def mean(data):
         return sum(data) / len(data)
>>>  def std(data):
         var = sum([(dp – mean(data))**2/(len(data) - 1) for dp in data])
         return math.sqrt(var)
>>>  def standardize(data):
         return [(dp – mean(data))/std(data) for dp in data]
>>>  def unstd(data, data_mean, data_std):
         return [(dp * data_std) + data_mean for dp in data]

>>> data = [3.5, 4.2, 1.2, 7.5, 2.2]
>>> norm_data = standardize(data)
>>> norm_data_mod = f(norm_data)

# But what if we want to go back to the original data?!
>>> mean_ = mean(data)
>>> std_  = std(data)
>>> modified_data = unstd(data, mean_, std_)
```

Standardization II: With functions

```
>>> data = [3.5, 4.2, 1.2, 7.5, 2.2]
>>> norm_data = standardize(data)
>>> norm_data_mod = f(norm_data)

# But what if we want to go back to the original data?!
>>> mean_, std_ = mean(data), std(data)
>>> modified_data = unstd(data, mean_, std_)

>>> data2 = [3.5, 4.2, 1.2, 7.5, 2.2]
>>> norm_data2 = standardize(data2)
>>> norm_data_mod2 = f(norm_data)

# But what if we want to go back to the original data?!
>>> mean2_ , std2_  = mean(data) , std(data)
>>> modified_data = unstd(data, mean_, std_)
```

The user has to save a assign and keep track of a lot of variables for every dataset!

Standardization II: With functions

Squidward says:

"functions are perfect"

```
>>> data = [3.5, 4.2, 1.2, 7.5, 2.2]
>>> norm_data = standardize(data)
>>> norm_data_mod = f(norm_data)

# But what if we want to go back to the original data?!
>>> mean_, std_ = mean(data), std(data)
>>> modified_data = unstd(data, mean_, std_)

>>> data2 = [3.5, 4.2, 1.2, 7.5, 2.2]
>>> norm_data2 = standardize(data2)
>>> norm_data_mod2 = f(norm_data)

# But what if we want to go back to the original data?!
>>> mean2_ , std2_  = mean(data) , std(data)
>>> modified_data = unstd(data, mean_, std_)
```

Professor Tentacles and his circle!

~~“functions are perfect”~~

Standardization III: With class

```
# mean function
>>>  class Std_scaler:
         """a standardization class."""”
```

Standardization III: With class

```
# mean function
>>>  class Std_scaler:
         """a standardization class."""
```

Docstrings are a great way to include documentation for your class

Standardization III: With class

```
# mean function
>>>  class Std_scaler:
         """a standardization class"""
         def _mean(self, data):
             return sum(data) / len(data)
         def _std(self, data):
             var = sum([(dp - mean(data))**2/(len(data) - 1) for dp in data])
             return math.sqrt(var)
```

Standardization III: With class

```
# mean function
>>>  class Std_scaler:
         """a standardization class"""
         def _mean(self, data):
             return sum(data) / len(data)
         def _std(self, data):
             var = sum([(dp - mean(data))**2/(len(data) - 1) for dp in data])
             return math.sqrt(var)
```

By convention methods internal to the class (not meant for the user to see) have a leading underscore.

Standardization III: With class

```
# mean function
>>>  class Std_scaler:
         """a standardization class"""
         def _mean(self, data):
             return sum(data) / len(data)
         def _std(self, data):
             var = sum([(dp - mean(data))**2/(len(data) - 1) for dp in data])
             return math.sqrt(var)
         def fit_transform(self, data):
             self.mean_, self.std_ = self._mean(data), self._std(data)
             return [(dp - self.mean_)/self.std_ for dp in data]
```

Standardization III: With class

```
# mean function
>>>  class Std_scaler:
        """a standardization class."""”
        def _mean(self, data):
            return sum(data) / len(data)
        def _std(self, data):
            var = sum([(dp – mean(data))**2/(len(data) - 1) for dp in data])
            return math.sqrt(var)
        def fit_transform(self, data):
            self.mean_, self.std_ = self._mean(data), self._std(data)
            return [(dp – self.mean_)/self.std_ for dp in data]
```

By convention in sklearn attributes that have been estimated from the data must always have a name ending with trailing underscore _

Standardization III: With class

```
# mean function
>>>  class Std_scaler:
         """a standardization class."""
         def _mean(self, data):
             return sum(data) / len(data)
         def _std(self, data):
             var = sum([(dp - mean(data))**2/(len(data) - 1) for dp in data])
             return math.sqrt(var)
         def fit_transform(self, data):
             self.mean_, self.std_ = self._mean(data), self._std(data)
             return [(dp - self.mean_)/self.std_ for dp in data]
         def inverse_transform(self, norm_data):
             return [dp * self.std_ + self.mean_ for dp in data]
```

Standardization III: With class

```
# mean function
>>>  class Std_scaler:
         """a standardization class."""
         def _mean(self, data):
             return sum(data) / len(data)
         def _std(self, data):
             var = sum([(dp - mean(data))**2/(len(data) - 1) for dp in data])
             return math.sqrt(var)
         def fit_transform(self, data):
             self.mean_, self.std_ = self._mean(data), self._std(data)
             return [(dp - self.mean_)/self.std_ for dp in data]
         def inverse_transform(self, norm_data):
             return [dp * self.std_ + self.mean_ for dp in data]
```

Enter the python class, superior to functions!

```
# mean function
>>>  class Std_scaler:
         """a standardization class."""
         […]

>>> data = [3.5, 4.2, 1.2, 7.5, 2.2]
>>> scaler = Std_scaler()
```

Enter the python class, superior to functions!

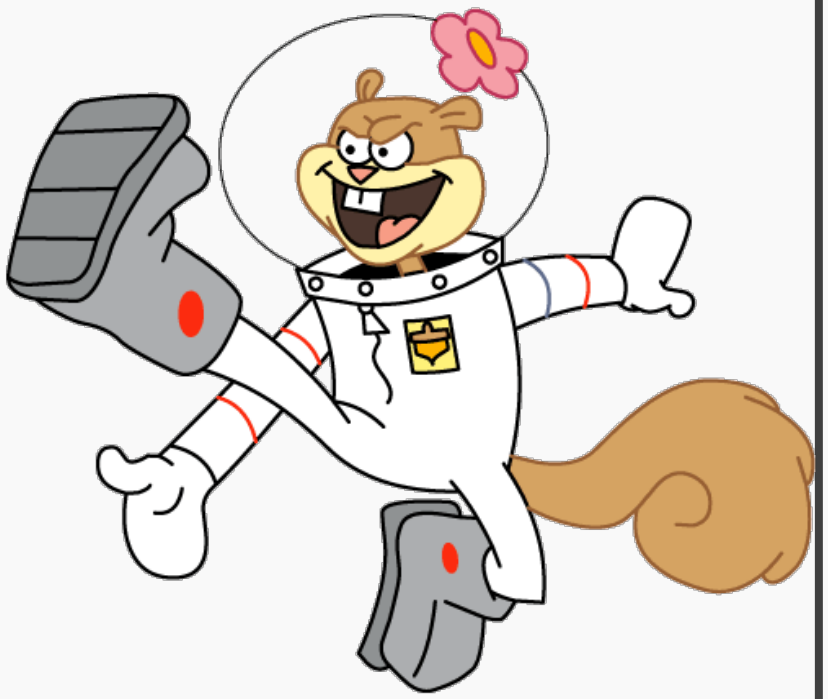
```
# mean function
>>>  class Std_scaler:
         """a standardization class."""
         […]

>>> data = [3.5, 4.2, 1.2, 7.5, 2.2]
>>> scaler = Std_scaler()
>>> norm_data = scaler.fit_transform(data)
```

Enter the python class, superior to functions!

```
# mean function
>>>  class Std_scaler:
         """a standardization class."""
         […]

>>> data = [3.5, 4.2, 1.2, 7.5, 2.2]
>>> scaler = Std_scaler()
>>> norm_data = scaler.fit_transform(data)
>>> altered_data = scaler.inverse_transform(f(norm_data))
```

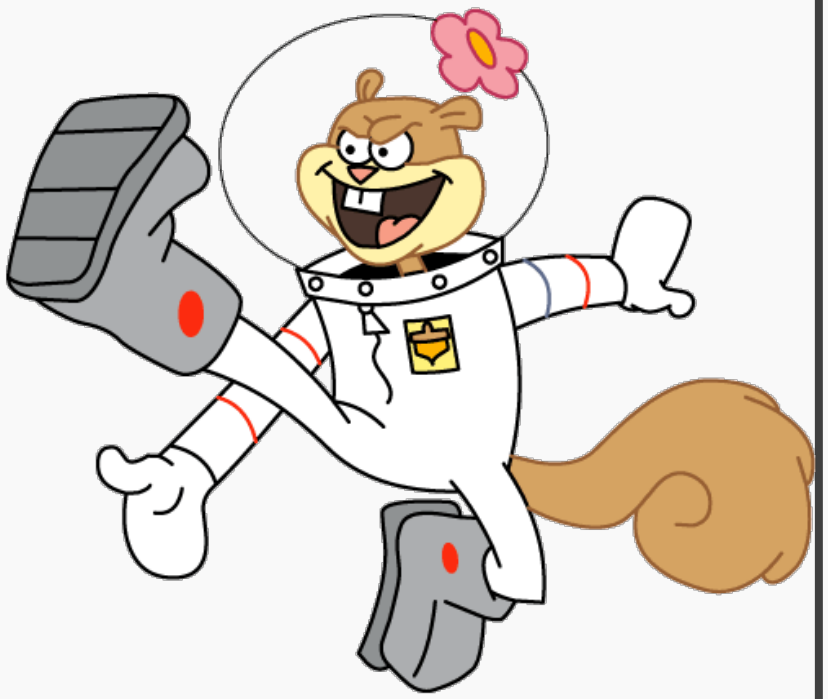
Enter the python class, superior to functions!

```
# mean function
>>>  class Std_scaler:
         """a standardization class."""
         […]

>>> data = [3.5, 4.2, 1.2, 7.5, 2.2]
>>> scaler = Std_scaler()
>>> norm_data = scaler.fit_transform(data)
>>> altered_data = scaler.inverse_transform(f(norm_data))
```



Enter the python class, superior to functions!

```
# mean function
>>>  class Std_scaler:
         """a standardization class."""
         […]

>>> data = [3.5, 4.2, 1.2, 7.5, 2.2]
>>> scaler = Std_scaler()
>>> norm_data = scaler.fit_transform(data)
>>> altered_data = scaler.inverse_transform(f(norm_data))


>>> data2 = [5.2, 0.2, 1.5, 6.3, 2.1]
>>> norm_data2 = scaler.fit_transform(data2)
>>> altered_data2 = scaler.inverse_transform(f(norm_data2))
```



Side by side: functions vs classes

```
>>> data = [3.5, 4.2, 1.2, 7.5, 2.2]
>>> data2 = [0.3, 3.9, 1.1, 2.5, 6.1]
```

```
>>> mean_ = mean(data)
>>> std_dev_ = std_dev(data)
>>> norm_data = standardize(data)
>>> altered_data = unstd(f(norm_data), mean_, std_dev_)

>>> mean2_ = mean(data2)
>>> std_dev2_ = std_dev(data2)
>>> norm_data2 = standardize(data2)
>>> altered_data2 = unstd(f(norm_data2), mean2_, std_dev2_)
```

```
>>> scaler = Std_scaler()
>>> norm_data = scaler.fit_transform(data)
>>> altered_data = scaler.inverse_transform(f(norm_data))

>>> norm_data = scaler.fit_transform(data2)
>>> altered_data2 = scaler.inverse_transform(f(norm_data2))
```

Side by side: functions vs classes

```
>>> data = [3.5, 4.2, 1.2, 7.5, 2.2]
>>> data2 = [0.3, 3.9, 1.1, 2.5, 6.1]
```

```
>>> mean_ = mean(data)
>>> std_dev_ = std_dev(data)
>>> norm_data = standardize(data)
>>> altered_data = unstd(f(norm_data), mean_, std_dev_)

>>> mean2_ = mean(data2)
>>> std_dev2_ = std_dev(data2)
>>> norm_data2 = standardize(data2)
>>> altered_data2 = unstd(f(norm_data2), mean2_, std_dev2_)
```

```
Characters: 288
Lines: 8
User defined objects: 4
User object calls: 8
```

```
>>> scaler = Std_scaler()
>>> norm_data = scaler.fit_transform(data)
>>> altered_data = scaler.inverse_transform(f(norm_data))

>>> norm_data = scaler.fit_transform(data2)
>>> altered_data2 = scaler.inverse_transform(f(norm_data2))
```

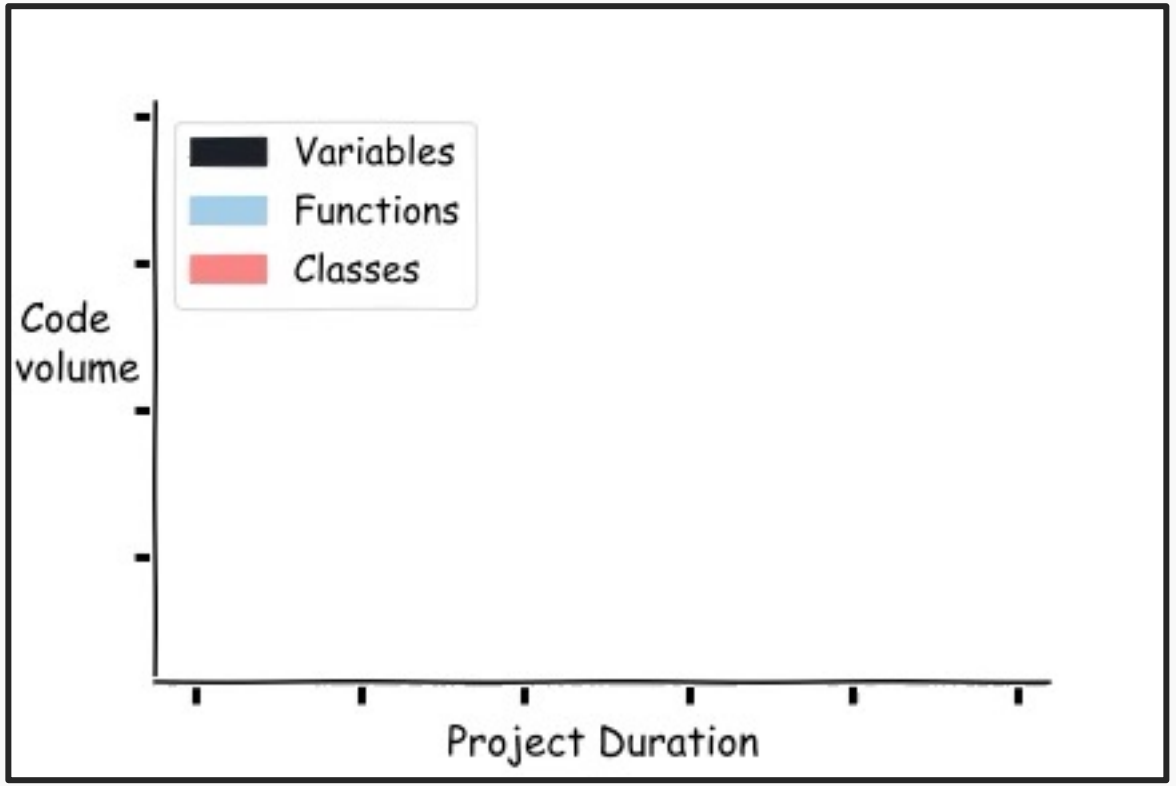
```
Characters: 226
Lines: 5
User defined objects: 1
User object calls: 5
```

Side by side comparison

```
Characters: 576
Lines: 16
Userdefined objects: 8
User object calls: 16
```

```
Characters: 288
Lines: 8
User defined objects: 4
User object calls: 8
```

```
Characters: 226
Lines: 5
User defined objects: 1
User object calls: 5
```



Efficiency of Data Science code:

More abstraction: *notebook* -> *functions* -> *classes*

Notebook only

Co2 per km: ∞

…

Efficiency of Data Science code:

More abstraction: *notebook* -> *functions* -> *classes*

Notebook only

Co2 per km: ∞

…

Efficiency of Data Science code:

More abstraction: *notebook* -> *functions* -> *classes*

Notebook only

Co2 per km: ∞

…

With Functions

Co2 per km: 250 g

Efficiency of Data Science code:

More abstraction: *notebook* -> *functions* -> *classes*

Notebook only

Co2 per km: ∞

With Functions

Co2 per km: 250 g

With Classes:

Co2 per km 0 g

Why should I care?

Data Science Applications

Examples

```
>>> from sklearn.preprocessing import StandardScaler
>>> data = [[0, 0], [0, 0], [1, 1], [1, 1]]
>>> scaler = StandardScaler()
>>> print(scaler.fit(data))
StandardScaler()
>>> print(scaler.mean_)
[0.5 0.5]
>>> print(scaler.transform(data))
[[-1. -1.]
 [-1. -1.]
 [ 1.  1.]
 [ 1.  1.]]
>>> print(scaler.transform([[2, 2]]))
[[3. 3.]]
```

Methods

`fit`(X[, y, sample_weight])	Compute the mean and std to be used for later scaling.
`fit_transform`(X[, y])	Fit to data, then transform it.
`get_params`([deep])	Get parameters for this estimator.
`inverse_transform`(X[, copy])	Scale back the data to the original representation
`partial_fit`(X[, y, sample_weight])	Online computation of mean and std on X for later scaling.
`set_params`(**params)	Set the parameters of this estimator.
`transform`(X[, copy])	Perform standardization by centering and scaling

Image source: https://www.ciat.org/fortaleciendo-el-maletin-de-herramientas-para-la-gestion-del-cumplimiento-tributario-machine-learning-1/?lang=en

- Python data science and Machine Learning (ML) packages are usually implemented as classes

Examples

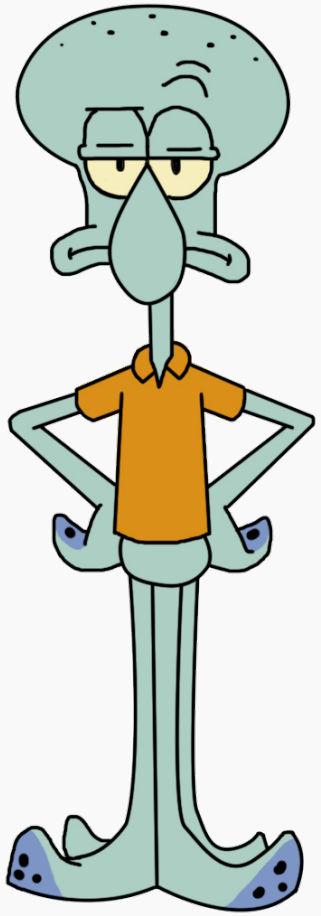
```
>>> from sklearn.preprocessing import StandardScaler
>>> data = [[0, 0], [0, 0], [1, 1], [1, 1]]
>>> scaler = StandardScaler()
>>> print(scaler.fit(data))
StandardScaler()
>>> print(scaler.mean_)
[0.5 0.5]
>>> print(scaler.transform(data))
[[-1. -1.]
 [-1. -1.]
 [ 1.  1.]
 [ 1.  1.]]
>>> print(scaler.transform([[2, 2]]))
[[3. 3.]]
```

Methods

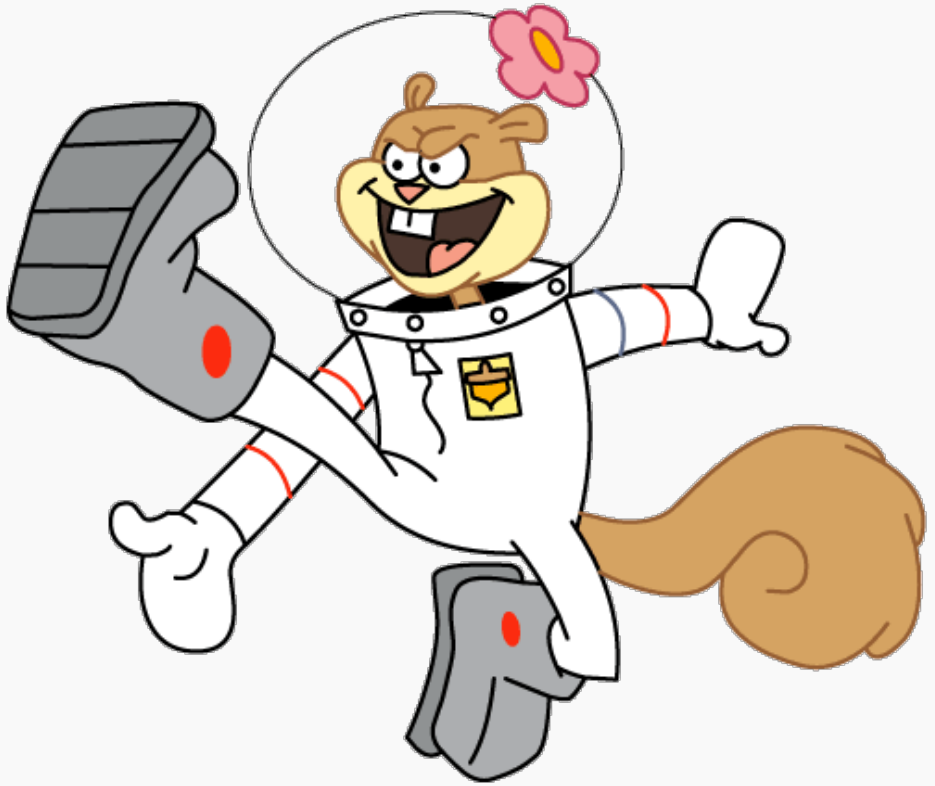
`fit`(X[, y, sample_weight])	Compute the mean and std to be used for later scaling.
`fit_transform`(X[, y])	Fit to data, then transform it.
`get_params`([deep])	Get parameters for this estimator.
`inverse_transform`(X[, copy])	Scale back the data to the original representation
`partial_fit`(X[, y, sample_weight])	Online computation of mean and std on X for later scaling.
`set_params`(**params)	Set the parameters of this estimator.
`transform`(X[, copy])	Perform standardization by centering and scaling

~~Functions bad!~~

~~I don't need classes~~



Classes are the basis of professional python for data science



Dunder Methods

Dunder methods I: initialization and callability

__init__, __call__

Dunder method

What is a dunder method?

1. Dunder means double underscore
2. Dunder methods ensure that objects follow a certain set of protocalls.

Object Initialization

What is __init__?

When python objects are created self is passed to the __init__ method and the object is initialized.

```
# Create a class template
class Model_S:
    def make(self):
        print(f"{self.year} Tesla Model S")
    def __init__(self, model_year):
```

Object Initialization

What is __init__?

When python objects are created self is passed to the __init__ method and the object is initialized.

We can then assign other attributes via other arguments passed to __init__.

```
# Create a class template
class Model_S:
    def make(self):
        print(f"{self.year} Tesla Model S")
    def __init__(self, model_year):
        self.year = model_year
```

Object Initialization

What is __init__?

When python objects are created self is passed to the __init__ method and the object is initialized.

We can then assign other attributes via other arguments passed to __init__. We can even call other methods from __init__!

```
# Create a class template
class Model_S:
    def make(self):
        print(f"{self.year} Tesla Model S")
    def __init__(self, model_year):
        self.year = model_year
        self.make()


# Create a class instance
>>> my_tesla = Model_S( model_year = 2017)
"2017 Tesla Model S”
```

Object Initialization

What is __init__?

When python objects are created self is passed to the __init__ method and the object is initialized.

We can then assign other attributes via other arguments passed to __init__. We can even call other methods from __init__!

```
# Create a class template
class Model_S:
    def make(self):
        print(f"{self.year} Tesla Model S")
    def __init__(self, model_year):
        self.year = model_year
        self.make()


# Create a class instance
>>> my_tesla = Model_S( model_year = 2017)
"2017 Tesla Model S”

# Instantiate the the class object
>>> your_tesla = Model_S( model_year = 2021)
"2021 Tesla Model S"
```

Object Callability

What is callability?

Callability is the ability of a python object to be called like a function.

```
# Create a class template
class Model_S:
    def __init__(self, loc = "the factory"):
        self.loc = loc
        self.where_am_I()
    def where_am_I(self):
        print(f"My Tesla is in {self.loc}")
```

Object Callability

What is callability?

Callability is the ability of a python object to be called like a function.

What happens if we don't have a call method defined?

```
# Create a class template
class Model_S:
    def __init__(self, loc = "the factory"):
        self.loc = loc
        self.where_am_I()
    def where_am_I(self):
        print(f"My Tesla is in {self.loc}")

# Create a class instance
>>> my_tesla = Model_S()
My Tesla is in the factory
>>> my_tesla()
TypeError: 'Tesla' object is not callable
```

Object Callability

What is callability?

Callability is the ability of a python object to be called like a function.

What happens if we don't have a call method defined?

You might have seen this error message before.

```
# Create a class template
class Model_S:
    def __init__(self, loc = "the factory"):
        self.loc = loc
        self.where_am_I()
    def where_am_I(self):
        print(f"My Tesla is in {self.loc}")

# Create a class instance
>>> my_tesla = Model_S()
My Tesla is in the factory
>>> my_tesla()
TypeError: 'Tesla' object is not callable

# make a list of cars
>>> cars = ["volvo", "nissan", "hyundai"]
>>> cars()
TypeError: 'list' object is not callable
```

Object Callability

What is __call__?

We can make python instances callable like normal functions via __call__.

```
class Tesla:
    def __init__(self, location = "the factory"):
        self.loc = location
        self.where_am_I()
    def where_am_I(self):
        print(f"My Tesla is in {self.loc}")
```

This can be very useful when an object needs to change state.

Object Callability

What is __call__?

We can make python instances callable like normal functions via __call__.

```
class Tesla:
    def __init__(self, location = "the factory"):
        self.loc = location
        self.where_am_I()
    def where_am_I(self):
        print(f"My Tesla is in {self.loc}")
    def __call__(self, location):
        self.loc = location
        self.where_am_I()
```

Object Callability

What is __call__?

We can make python instances callable like normal functions via __call__.

```
class Tesla:
    def __init__(self, location = "the factory"):
        self.loc = location
        self.where_am_I()
    def where_am_I(self):
        print(f"My Tesla is in {self.loc}")
    def __call__(self, location):
        self.loc = location
        self.where_am_I()

# Create a class instance
>>> my_tesla = Tesla()
 "My Tesla is in the factory”
```

Object Callability

What is __call__?

We can make python instances callable like normal functions via __call__.

```
class Tesla:
    def __init__(self, location = "the factory"):
        self.loc = location
        self.where_am_I()
    def where_am_I(self):
        print(f"My Tesla is in {self.loc}")
    def __call__(self, location):
        self.loc = location
        self.where_am_I()

# Create a class instance
>>> my_tesla = Tesla()
 "My Tesla is in the factory"

# treat our class instance like a function
>>> my_tesla("California")
"My Tesla is in California”
```

Object Callability

What is __call__?

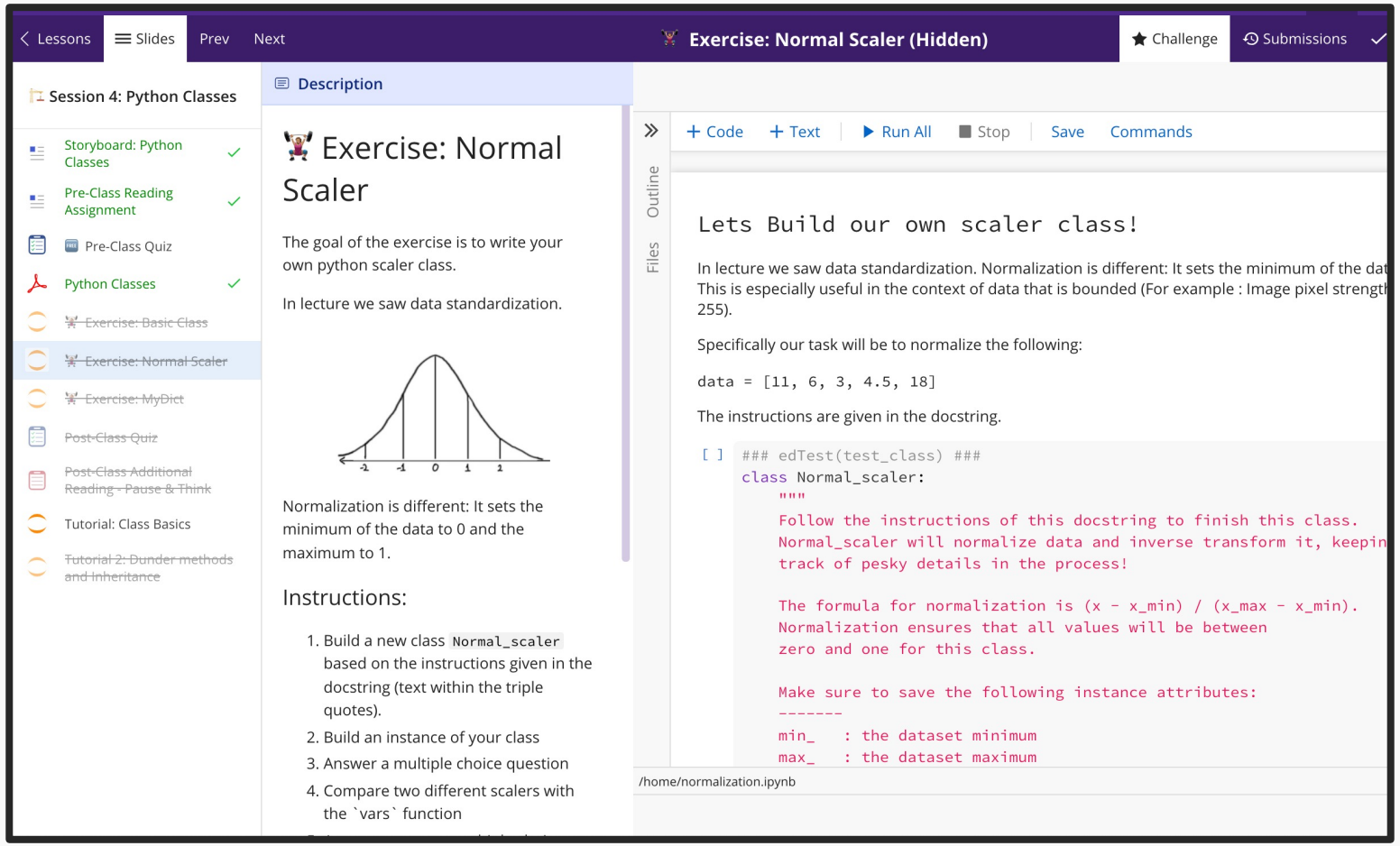
We can make python instances callable like normal functions via __call__.

```
class Tesla:
    def __init__(self, location = "the factory"):
        self.loc = location
        self.where_am_I()
    def where_am_I(self):
        print(f"My Tesla is in {self.loc}")
    def __call__(self, location):
        self.loc = location
        self.where_am_I()

# Create a class instance
>>> my_tesla = Tesla()
 "My Tesla is in the factory"

# treat our class instance like a function
>>> my_tesla("California")
"My Tesla is in California"

>>> my_tesla.loc
"California"
```

We can even return a value using the __call__() magic method

Exercise II, Build your own scaler!

__init__, __call__



Dunder methods II: (iterables)
__getitem__ , __setitem__ , __len__

__getitem__ and __setitem__

```
from random import randrange as rand

class Harlist:
    def __init__(self, num = 5):
        self.list = [rand(1, num, 1) for _ in
range(num)]
```

[…]

__getitem__ and __setitem__

```
from random import randrange as rand

class Harlist:
    def __init__(self, num = 5):
            self.list = [rand(1, num, 1) for _ in
range(num)]

# Create a class instance
>>> random_ints = Harlist()
>>> random_ints.list
[5, 3, 4, 1, 4]
```

[…]

__getitem__ and __setitem__

```
from random import randrange as rand

class Harlist:
    def __init__(self, num = 5):
            self.list = [rand(1, num, 1) for _ in
range(num)]

# Create a class instance
>>> random_ints = Harlist()
>>> random_ints.list
[5, 3, 4, 1, 4]
>>> random_ints[0]
TypeError: 'Rand_Int_List' object is not
subscriptable
```

[…]

__getitem__ and __setitem__

Unless we implement __getitem__ and __setitem__ we can’t use the index operator[]

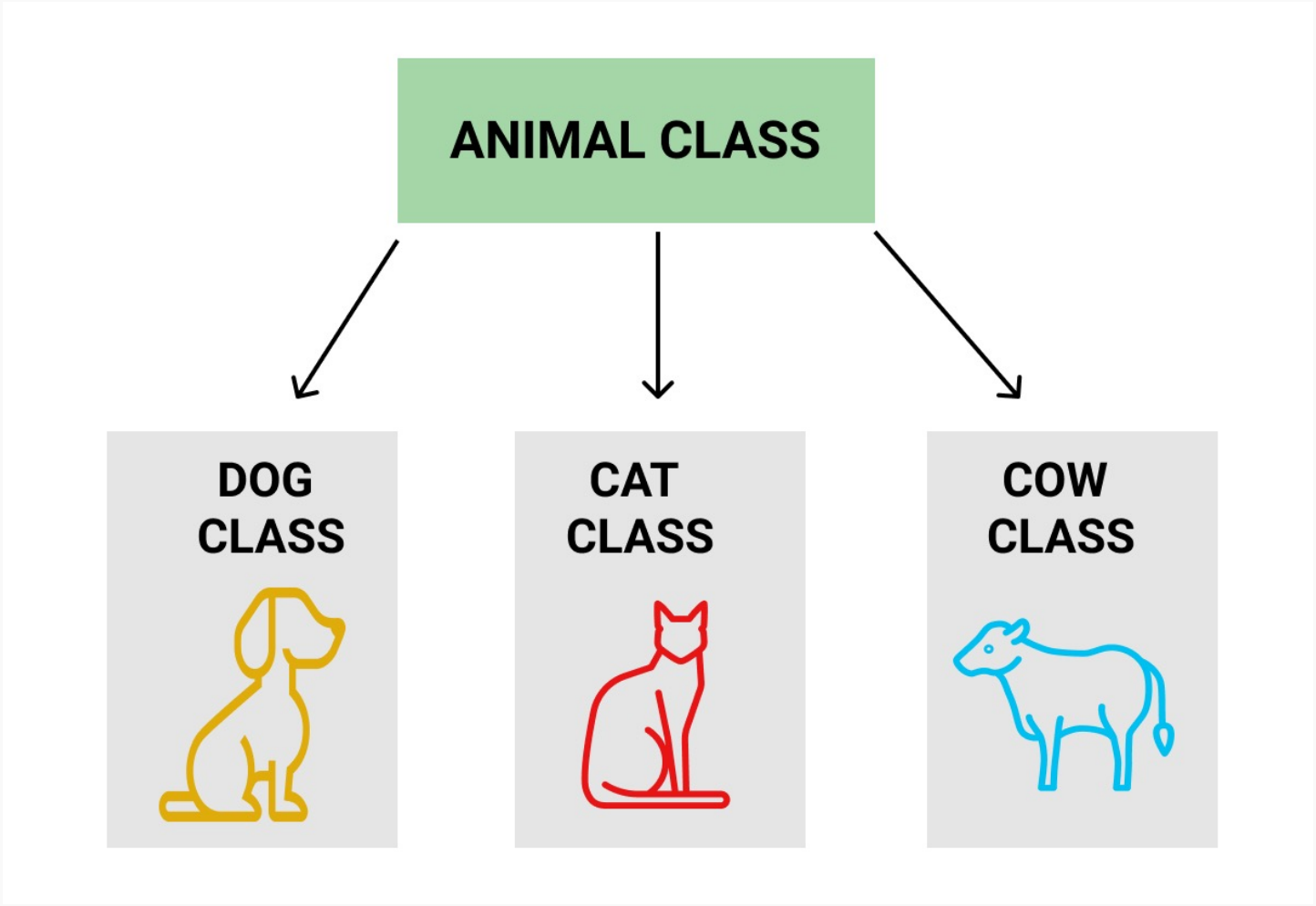
```
from random import randrange as rand

class Harlist:
    def __init__(self, num = 5):
            self.list = [rand(1, num, 1) for _ in
range(num)]

# Create a class instance
>>> random_ints = Harlist()
>>> random_ints.list
[5, 3, 4, 1, 4]
>>> random_ints[0]
TypeError: 'Rand_Int_List' object is not
subscriptable
```

[…]

Act III Inheritance

Act III Inheritance

In **object-oriented** programming, **inheritance** is the mechanism of basing an object or class upon another object (prototype-based **inheritance**) or class (class-based **inheritance**), retaining similar implementation. …
An **inherited** class is called a subclass of its parent class or super class.


ANIMAL CLASS
DOG CLASS
CAT CLASS
COW CLASS

__getitem__, __setitem__ and Inheritance

```
from random import randrange as rand

class Harlist_child(Harlist):
    def __getitem__(self, key):
        return self.list[key]
```

[…]

__getitem__, __setitem__ and Inheritance

```
from random import randrange as rand

class Harlist_child(Harlist):
    def __getitem__(self, key):
        return self.list[key]
```

[…]

__getitem__, __setitem__ and Inheritance

```
from random import randrange as rand

class Harlist_child(Harlist):
    def __getitem__(self, key):
        return self.list[key]
```

Here we have snuck in the first example of inheritance. We don't need to re-write the __init__ function from the previous class, we can inherit it!

```
class Harlist:
    def __init__(self, num):
        self.list = [rand(1, 6, 1) for _ in
range(num)]
```

[…]

__getitem__, __setitem__ and Inheritance

```
from random import randrange as rand

class Harlist_child(Harlist):
    def __getitem__(self, key):
        return self.list[key]
    def __setitem__(self, index, value):
        self.list[index] = value
```

[…]

__getitem__, __setitem__

```
from random import randrange as rand

class Harlist_child(Harlist):
    def __getitem__(self, key):
        return self.list[key]
    def __setitem__(self, index, value):
        self.list[index] = value

# Create a class instance
>>> random_ints = Harlist_child()
>>> random_ints.list
[4, 3, 4, 2, 4]
>>> random_ints[0]
4
```

[…]

__getitem__, __setitem__

Nice! Now it works as expected.

```
from random import randrange as rand

class Harlist_child(Harlist):
    def __getitem__(self, key):
        return self.list[key]
    def __setitem__(self, index, value):
        self.list[index] = value

# Create a class instance
>>> random_ints = Harlist_child()
>>> random_ints.list
[4, 3, 4, 2, 4]
>>> random_ints[0]
4
>>> random_ints[0] = 100
>>> random_ints[0]
100
```

[…]

__getitem__, __setitem__

But we can't print the list!

```
from random import randrange as rand

class Harlist_child(Harlist):
    def __getitem__(self, key):
        return self.list[key]
    def __setitem__(self, index, value):
        self.list[index] = value

# Create a class instance
>>> random_ints = Harlist_child()
>>> random_ints.list
[4, 3, 4, 2, 4]
>>> random_ints[0]
4
>>> random_ints[0] = 100
>>> random_ints[0]
100
>>> print(random_int)
<__main__.Rand_Int_List object at 0x7fb8d029b700>
```

[…]

Dunder methods III: __str__, __repr__

Object Representation

What are __str__ and __repr__?

__repr__ is formal string representation.

```
class Tesla:
    def __init__(self, model, year = 2021):
        self.year, self.model  = year, model
    def __repr__(self):
        print("repr scope")
        str = f"{self.year} Model {self.model}"
        return str
```

Object Representation

What are __str__ and __repr__?

__repr__ is formal string representation.

__str__ is informal string representation.

```
class Tesla:
    def __init__(self, model, year = 2021):
        self.year, self.model  = year, model
    def __repr__(self):
        print("repr scope")
        str = f"{self.year} Model {self.model}"
        return str
    def __str__(self):
        print("str scope")
        str = f"{self.year} Model {self.model}"
        return str
```

Object Representation

What are __str__ and __repr__?

__repr__ is formal string representation.

__str__ is informal string representation.

```
class Tesla:
    def __init__(self, model, year = 2021):
        self.year, self.model  = year, model
    def __repr__(self):
        print("repr scope")
        str = f"{self.year} Model {self.model}"
        return str
    def __str__(self):
        print("str scope")
        str = f"{self.year} Model {self.model}"
        return str

# Create a class instance
>>> my_tesla = Tesla("X")
>>> my_tesla
repr scope
2021 Model X
```

Object Representation

What are __str__ and __repr__?

__repr__ is formal string representation.

__str__ is informal string representation.

```
class Tesla:
    def __init__(self, model, year = 2021):
        self.year, self.model  = year, model
    def __repr__(self):
        print("repr scope")
        str = f"{self.year} Model {self.model}"
        return str
    def __str__(self):
        print(”str scope")
        str = f"{self.year} Model {self.model}"
        return str

# Create a class instance
>>> my_tesla = Tesla("X")
>>> my_tesla
repr scope
2021 Model X

>>> print(my_tesla)
str scope
2021 Model X
```

Object Representation

What are __str__ and __repr__?
__repr__ is formal string representation.
__str__ is informal string representation.

When we lack __str__ python falls back to __repr__ (and not the other way around), so __repr__ is more important.

```
class Tesla:
    def __init__(self, model, year = 2021):
        self.year, self.model  = year, model
    def __repr__(self):
        print("repr scope")
        str = f"{self.year} Model {self.model}"
        return str

# Create a class instance
>>> my_tesla = Tesla("X")
>>> my_tesla
repr scope
2021 Model X

>>> print(my_tesla)
repr scope
2021 Model X
```

Object Representation

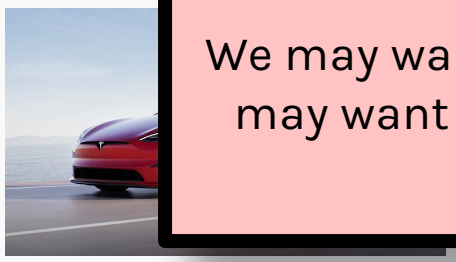
What are __str__ and __repr__?

__repr__ is formal string representation.

__str__ is informal string representation.

When we lack __str__ python falls back to __repr__ (and not the other way around), so __repr__ is more important.

```
class Tesla:
    def __init__(self, model, year = 2021):
        self.year, self.model  = year, model
    def __repr__(self):
        print("repr scope")
        str = f"{self.year} Model {self.model}"
        return str

# Create a class instance
>>> my_tesla = Tesla("X")
>>> my_tesla
repr scope
2021 Model X

>>> print(my_tesla)
repr scope
2021 Model X
```



We may want to customize the way we can interact with object representation. For example we may want more information from print and a simpler version from repr, allowing us to more easily interact with our machine learning models

Capping off Dunder Functions:

```
from random import randrange as rand

class Harlist_subsubclass(Harlist_subclass):
  […]
```

[…]

Capping off Dunder Functions:

```
from random import randrange as rand

class Harlist_subsubclass(Harlist_subclass):
  def __len__(self):
      return len(self.list)
```

[…]

Capping off Dunder Functions:

```
from random import randrange as rand

class Harlist_subsubclass(Harlist_subclass):
  def __len__(self):
      return len(self.list)
  def __repr__(self):
      return str(self.list)
```

[…]

Capping off Dunder Functions:

```
from random import randrange as rand

class Harlist_subsubclass(Harlist_subclass):
  def __len__(self):
      return len(self.list)
  def __repr__(self):
      return str(self.list)

# Create a class instance
>>> random_ints = Harlist_subsubclass(3)
>>> print(random_ints)
[4, 0 ,5]
>>> random_ints, len(random_ints)
[4, 0 ,5], 3
```

[…]

Object Oriented Programming (OOP): the four pillars

I. Encapsulation

II. Abstraction

III. Inheritance

IV. Polymorphism

Exercise to implement your own dictionary

Lets say we are trying to make the following assignment:

my_dict[“bye”] = ”hasta luego”

Exercise to implement your own dictionary

You might be surprised but a dictionary is just a list of lists!

We initialize the list of lists like so, up to n long.

storage = [[], [], [], [], ...]

Exercise to implement your own dictionary

You might be surprised but a dictionary is just a list of lists!

We initialize the list of lists like so, up to n long.

storage = [[], [], [], [], ...]

Then we use a hash function to convert the key to an integer.

For example:

hash(123) = 123

hash("bye") = -58445416447054340 20

Exercise to implement your own dictionary

Next we take the modulo of these hashes by the length of the storage. hash("bye") % 1000 = -5844541644705434020 = 980

Exercise to implement your own dictionary

Next we take the modulo of these hashes by the length of the storage. For example my_dict[“bye”] = ”hasta luego”

hash("bye") % 1000 = -5844541644705434020 = 980

So now index 998 of the list will look like this:

[…, [[“bye”, “hasta luego”]], …]

The final pillar: Polymorphism!

Polymorphism is **the ability of an object to take on many forms**.

```
from random import randrange as rand

class Harlist_squared(Harlist_subclass):
  def __init__(self, num):
        self.list = [rand(1, num, 1)**2 for _ in
range(num)]
```

Polymorphism: overwriting parent methods

Here we overwrote the upstream classes method.

```
from random import randrange as rand

class Harlist_squared(Harlist_subclass):
  def __init__(self, num):
        self.list = [rand(1, num, 1)**2 for _ in
range(num)]
```

Polymorphism: overwriting parent methods

```
from random import randrange as rand

class Harlist_squared(Harlist_subclass):
  def __init__(self):
        self.list = [rand(1, 6, 1)**2 for _ in
range(num)]



# Create a class instance
>>> square_ints = Harlist_squared(5)
>>> square_ints, len(square_ints)
[1, 16, 9, 4, 9], 5
```

Polymorphism: overwriting parent methods

We can overwrite methods from the parent class as shown

```
from random import randrange as rand

class Harlist_squared(Harlist_subclass):
  def __init__(self):
        self.list = [rand(1, 6, 1)**2 for _ in
range(num)]



# Create a class instance
>>> square_ints = Harlist_squared(5)
>>> square_ints, len(square_ints)
[1, 16, 9, 4, 9], 5
```

Polymorphism: super()

Alternatively we can use the super() function to call methods from the parent class.

```
from random import randrange as rand

class Harlist_squared(Harlist_subclass):
  def __init__(self, num):
        self.list = [rand(1, num, 1)**2 for _ in
range(num)]
  def init_parent(self, num):
        super().__init__(num)
```

Polymorphism: super()

Alternatively we can use the super() function to call methods from the parent class.

Think of super as a place holder for the parent class.

```
from random import randrange as rand

class Harlist_squared(Harlist_subclass):
  def __init__(self, num):
        self.list = [rand(1, num, 1)**2 for _ in
range(num)]
  def init_parent(self, num):
        super().__init__(num)


# Create a class instance
>>> square_ints = Harlist_squared(5)
>>> square_ints, len(square_ints)
[1, 16, 9, 4, 9], 5
```

Polymorphism: super()

Alternatively we can use the super() function to call methods from the parent class.

Think of super as a place holder for the parent class.

Here we called the parent classes __init__ method!

```
from random import randrange as rand

class Harlist_squared(Harlist_subclass):
  def __init__(self):
        self.list = [rand(1, 6, 1)**2 for _ in
range(num)]
  def init_parent(self, num):
        super().__init__(num)


# Create a class instance
>>> square_ints = Harlist_squared(5)
>>> square_ints, len(square_ints)
[1, 16, 9, 4, 9], 5

>>> square_ints.init_parent(5)
>>> square_ints
[3, 5, 2, 2, 4]
```

Polymorphism: super()

Think of it this way:

super() ==
[Parent_class]

```
from random import randrange as rand

class Harlist_squared(Harlist_subclass):
  def __init__(self):
        self.list = [rand(1, 6, 1)**2 for _ in
range(num)]
  def init_parent(self, num):
        Parent = super()
        Parent.__init__(num)

# Create a class instance
>>> square_ints = Harlist_squared(5)
>>> square_ints, len(square_ints)
[1, 16, 9, 4, 9], 5

>>> square_ints.init_parent(5)
>>> square_ints
[3, 5, 2, 2, 4]
```

Polymorphism: super()

Think of it this way:

Super() == [Parent_class]

Dunder methods ensure that objects follow a certain set of protocalls. This is a form of polymorphism.

```
from random import randrange as rand

class Harlist_squared(Harlist_subclass):
  def __init__(self):
        self.list = [rand(1, 6, 1)**2 for _ in
range(num)]
  def init_parent(self, num):
        Parent = super()
        Parent.__init__(num)

# Create a class instance
>>> square_ints = Harlist_squared(5)
>>> square_ints, len(square_ints)
[1, 16, 9, 4, 9], 5

>>> square_ints.init_parent(5)
>>> square_ints
[3, 5, 2, 2, 4]
```

Key	Value
	Brownie
	Simba
	Gillu
	Sandy

Key Hashing

MyDict()

```
[
[['Dog','Brownie'],
[ ],
[ ],
[ ],
[ ],
[ ],
[['Cat','Simba'],['Camel','Sandy']],
[ ],
[ ],
[ ],
[ ],
['Hamster','Gillu'],
[ ],
[ ],
[ ],
[ ],...
]
```

~~What is a python class ?~~

What is a docstring?

What is __init__?

What is staticmethod ?

What is Inheritance?

~~What is self ?~~

What is __call__ ?

What is __str__?

~~What is the difference between classes and methods?~~

What is a dunder Method?

```python
class Cookie:
    """A generic cookie class"""
    calories_per_gram  = {"sugar" : 4, "butter" : 7,
                          "cornstarch" : 4, "egg" : 2,
                         "chocolate_chips" : 5}

    def __init__(self, cookie_type, sugar_grams):
        self.cookie_type,  self.sugar_grams = cookie_type,  sugar_grams

    @staticmethod
    def calc_cal(ingredient_dict, calories_per_gram):
        total_calories = 0
        for ingredient, amount in ingredient_dict.items():
            total_calories += amount * calories_per_gram[ingredient]
        return total_calories

class Chocolate_chip(Cookie):
    """A chocolate chip cookie recipe!"""
    def __init__(self, num_chocolate_chips):
        self.cookie_type = "chocolate_chip"
        self.num_choc_chips = num_chocolate_chips
        #grams
        self.ingredient_dict = { "sugar" : 32, "butter" : 170,
                                 "chocolate_chips" : num_chocolate_chips,
                                 "cornstarch" : 12, "egg" : 50
                                }
        self.num_cookies = 16

    def __str__(self):
        total_calories = self.calc_cal(self.ingredient_dict, self.calories_per_gram)
        cal_per_cookie = round(total_calories/self.num_cookies)
        cchips = self.ingredient_dict
        return 'I am a {} with {} calories and {} chocolate chips'.format(
                self.cookie_type, cal_per_cookie, self.num_choc_chips)
```