Classes and OOP

Intro

Classes allow programmers to define NEW TYPES!

Recall "OLD" Data Types:

- List
- Tuple
- int
- float
- str
- bool
- set
- dict

Classes allow programmers to define NEW TYPES!

- Classes allow programmers to define new types!
- Examples
    - class Day():
    - class Person():
    - class Train():
- The ^^^ would create the data type Day, and Person

- Examples creating class instance
- class Train():
    - need instructions about how to construct an instance
    - need methods managing data about an instance
    - need dunder methods to help us print the object

Class Basics

- Classes keep code organized and modular
  - Classes store data
  - Classes store methods (functions) that operate on the data
  - outside of the class implementation, the class is abstract - it hides details from the rest of the code

- Examples creating class instance and using methods
- `# create a Profiler object to time code segments`
- profiler = Profiler()
- profiler.start() `# use the start method defined for a Profiler`
- profiler.stop() `# use the stop method defined for a Profiler`

- nb, you can use start() and stop() without knowing the implementation
- sounds a little related to the abstraction that functions provide…

Comparison of Functions and Classes

- Functions
  - `def` keyword
  - annotations, inputs, implementation, return statement
  - functions get "called"
  - functions operate on input data
  - function can be used based on specifications

- Classes
  - `class` keyword
  - **constructors, attributes, methods**
  - the constructor creates the object using special __init__ function
  - classes get "**instantiated**"
  - class methods operate on data stored in an **instance** of the class
  - class methods are not normally accessible to other objects

Everything in a class can and must be customized!

- The computer will not know what to do if it doesn't have instructions
- Everything has to be defined in a class
  - comparison instructions (>, <, >=, <=)
  - equality instructions (==)
  - addition instructions (+)
  - searching/look-up methods (in)
  - printing/representation instructions (str, repr)

Terms - Classes contain:

- attributes
  - values or data associated with an object
  - accessed with . notation
  - not callable
- methods
  - function that operates on the object (and it's attributes ^^^)
  - accessed with . notation
  - callable!

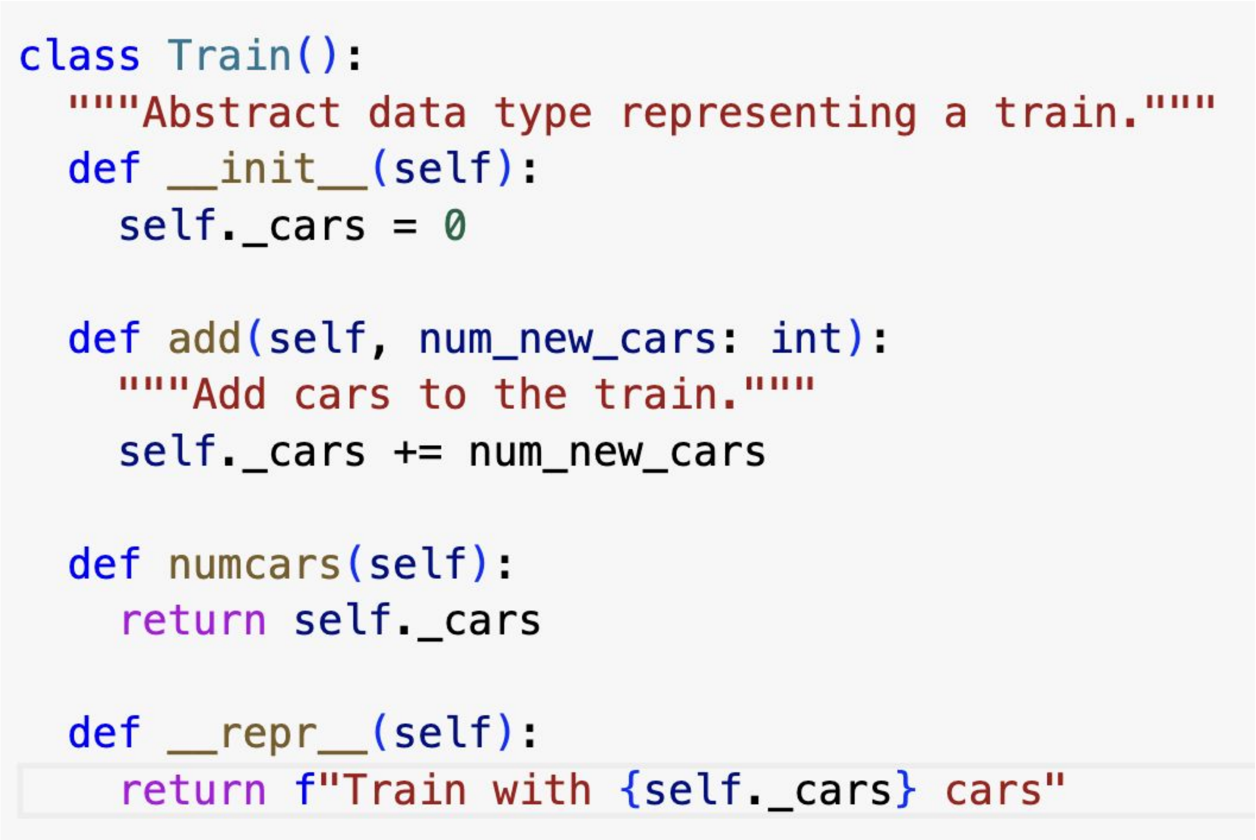
Terms - Classes contain:

- magic methods or dunder methods
    - functions defined with a special format, special names that the interpreter ALREADY KNOWS
    - __repr__ gets called AUTOMATICALLY in print statements
    - __init__ gets called AUTOMATICALLY to instantiate or create an object

Example Class

Notice:

- keyword
- __init__()
- add()
- numcars()
- __repr__()
- self?

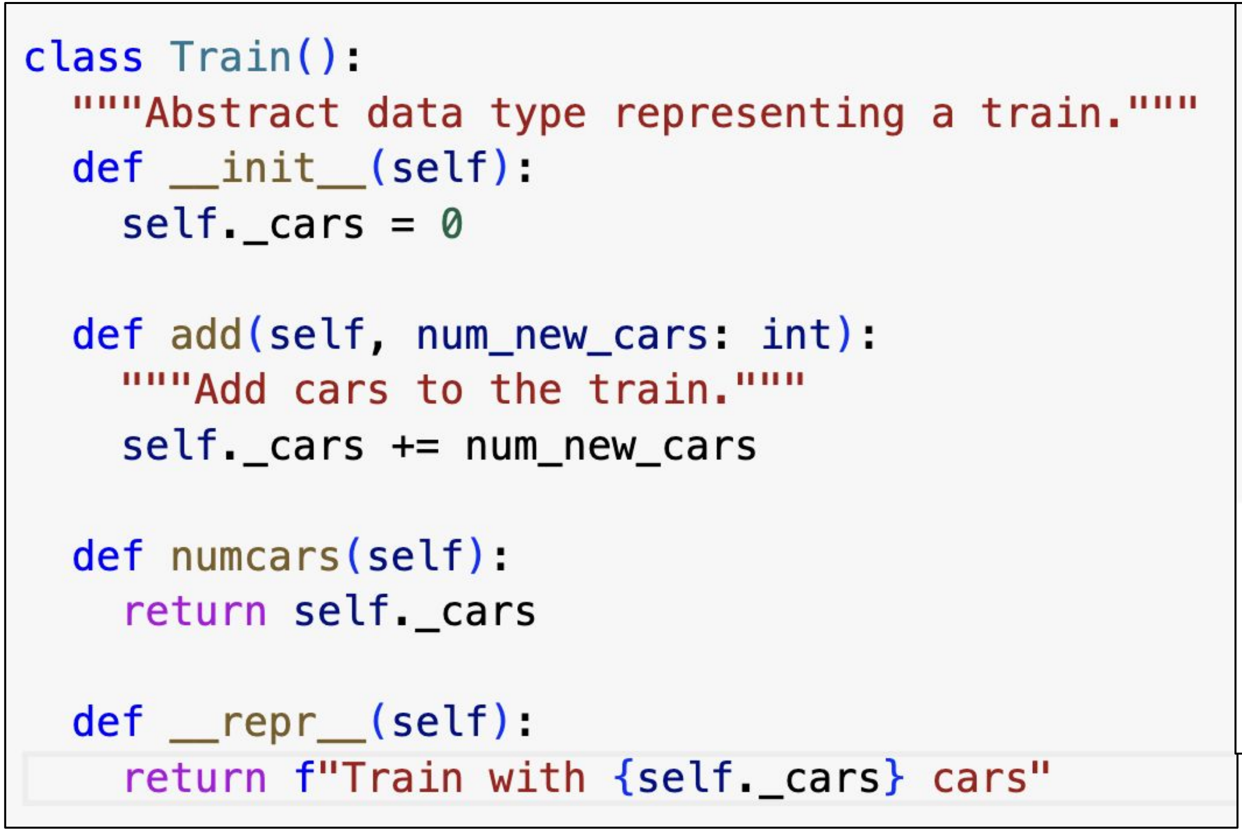


```python
class Train():
  """Abstract data type representing a train."""
  def __init__(self):
    self._cars = 0

  def add(self, num_new_cars: int):
    """Add cars to the train."""
    self._cars += num_new_cars

  def numcars(self):
    return self._cars

  def __repr__(self):
    return f"Train with {self._cars} cars"
```

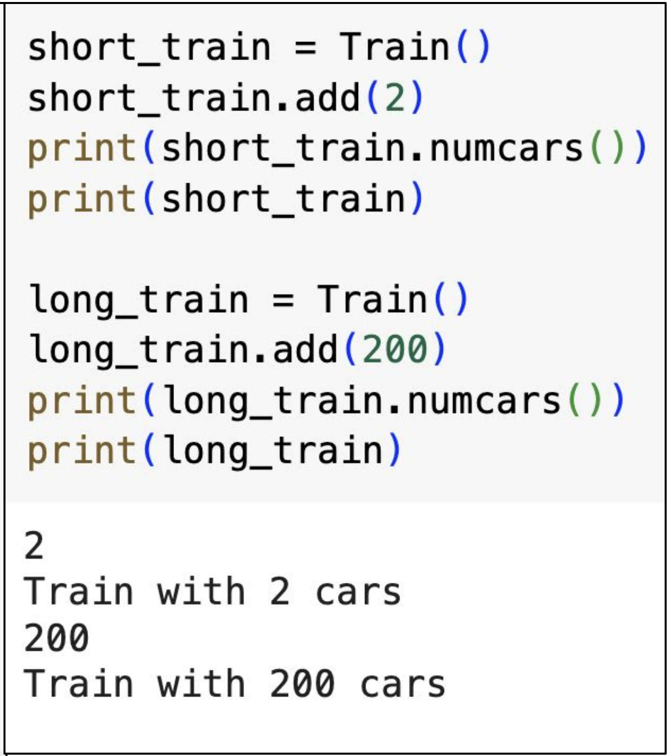
Terms - conventions

- `self`
  - `self` is the conventional name given to the first formal parameter in class methods
  - when method is called, self refers to the instantiated object itself
  - when method is called, self can be skipped
  - "The object associated with the expression preceding the dot is **implicitly** passed as the first parameter to the method"

Example Class



```python
class Train():
  """Abstract data type representing a train."""
  def __init__(self):
    self._cars = 0

  def add(self, num_new_cars: int):
    """Add cars to the train."""
    self._cars += num_new_cars

  def numcars(self):
    return self._cars

  def __repr__(self):
    return f"Train with {self._cars} cars"
```

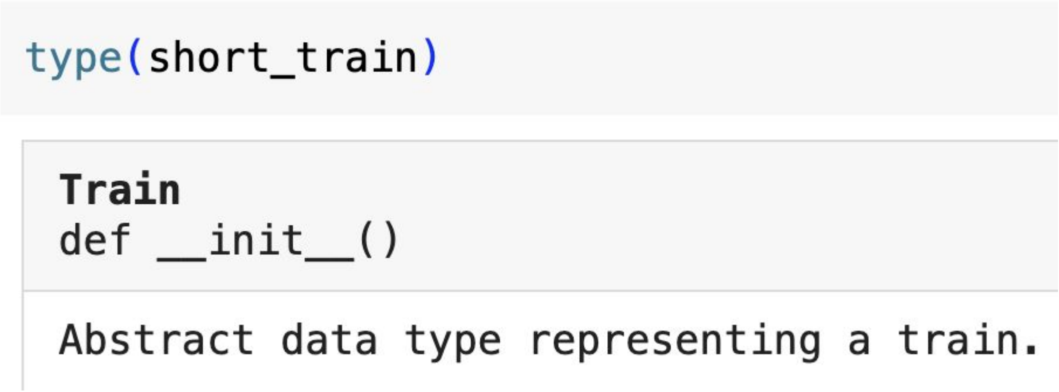


```python
short_train = Train()
short_train.add(2)
print(short_train.numcars())
print(short_train)

long_train = Train()
long_train.add(200)
print(long_train.numcars())
print(long_train)
```

```
2
Train with 2 cars
200
Train with 200 cars
```

Terms

- instance
  - instance refers to a created, named object that is of type CLASS
- class
  - the ABSTRACT data type, not the instance!

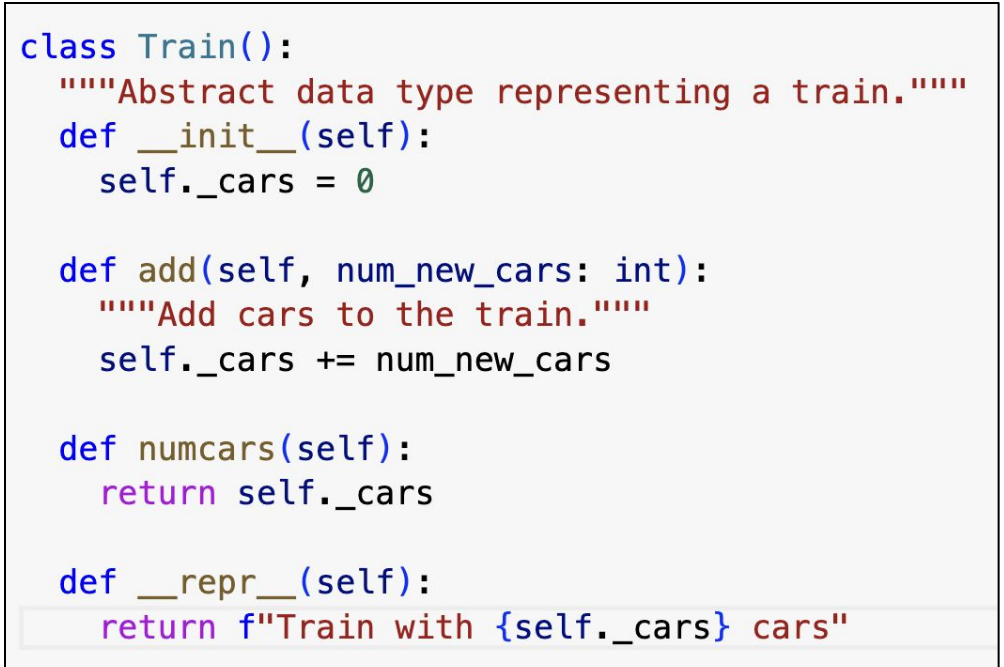


```
type(short_train)
```

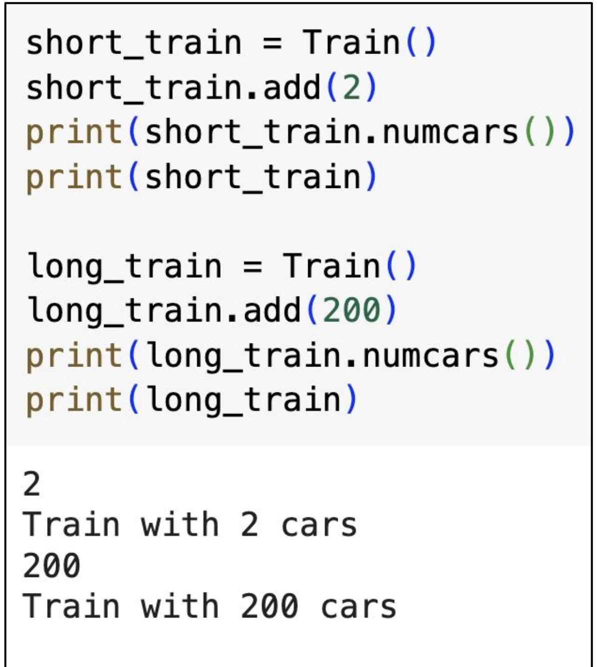
```
Train
def __init__()

Abstract data type representing a train.
```

Other conventions

- "_" this means private, for use inside the class only
- don't access instance variables that start with "_"



```python
class Train():
  """Abstract data type representing a train."""
  def __init__(self):
    self._cars = 0

  def add(self, num_new_cars: int):
    """Add cars to the train."""
    self._cars += num_new_cars

  def numcars(self):
    return self._cars

  def __repr__(self):
    return f"Train with {self._cars} cars"
```



```python
short_train = Train()
short_train.add(2)
print(short_train.numcars())
print(short_train)

long_train = Train()
long_train.add(200)
print(long_train.numcars())
print(long_train)
```

```
2
Train with 2 cars
200
Train with 200 cars
```

Google Form

https://forms.gle/V1rkvgmPxEoCACkL8

Read Chapter 10, and other references

- ref: https://stackoverflow.com/questions/46312470/difference-between-methods-and-attributes-in-python
- ref: https://towardsdatascience.com/practical-python-class-vs-instance-variables-431fd16430d

Overloading operators

- +: __add__
- -: __sub__
- **: __pow__
- <<: __lshift__
- *: __mul__
- /: __truediv__
- //: __floordiv__
- %: __mod__
- |: __or__
- <: __lt__

- ∧: __xor__
- >: __gt__
- >>: __rshsift__
- ==: __eq__
- <=: __le__
- &: __and__
- !=: __ne__
- >=: __ge__
- str: __str__

- len: __len__
- hash: __hash__
- repr: __repr__ - https://stackoverflow.com/questions/1436703/what-is-the-difference-between-str-and-repr