FINDING MAGIC IN PYTHON

Anna-Lena Popkes

July 29, 2021

ABOUT ME

Machine Learning Engineer @inovex

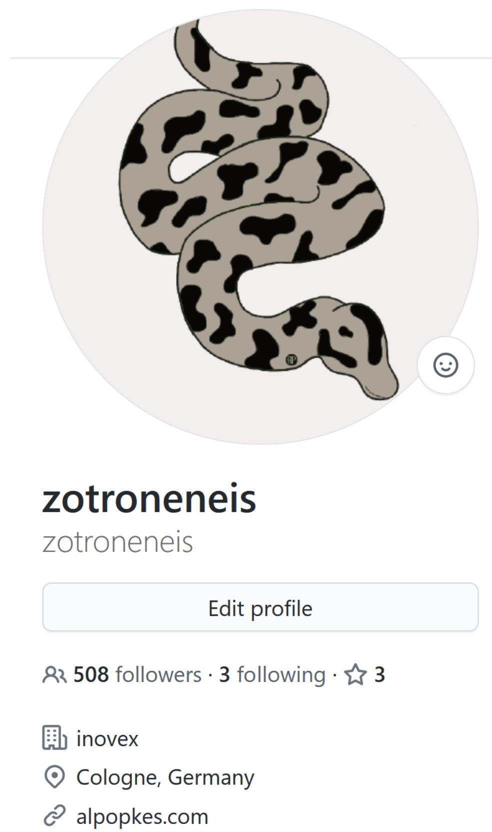
inovex

ABOUT ME

Member of "KI Macht Schule"

ABOUT ME

- Github: zotroneneis
- Personal webpage



ABOUT ME

Github: zotroneneis

machine_learning_basics

Plain python implementations of basic machine learning algorithms

Jupyter Notebook ☆ 3.5k 685

magical_universe

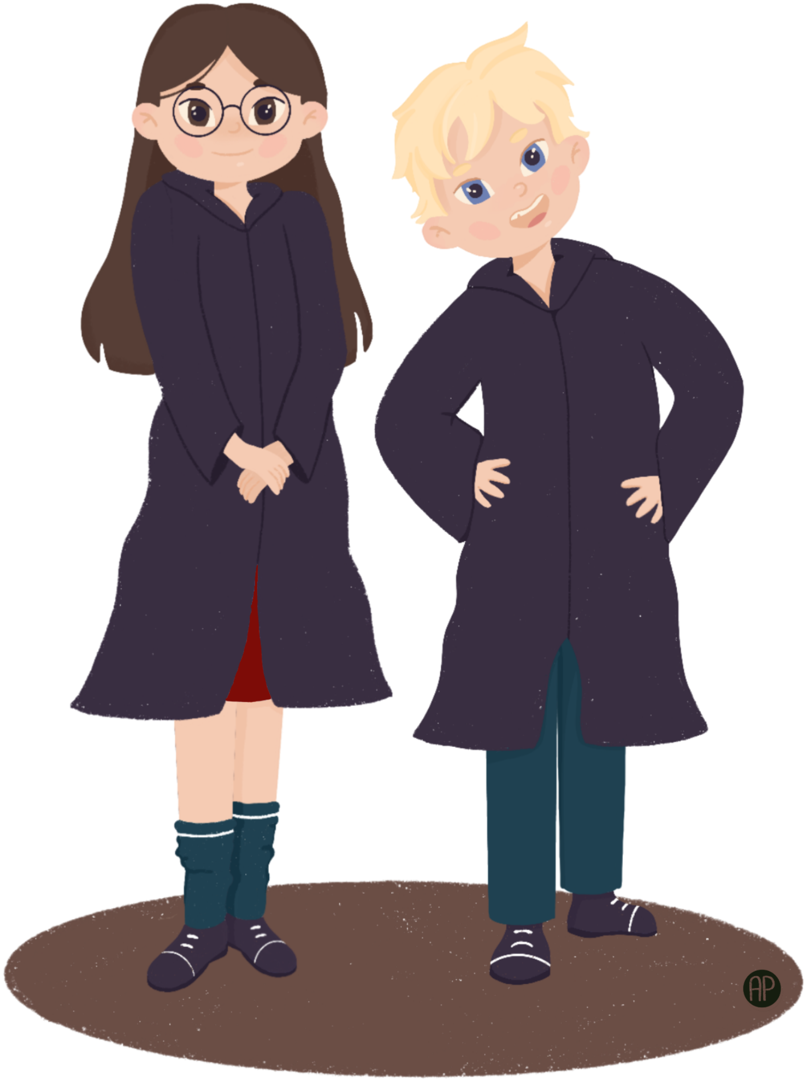
Awesome Python features explained using the world of magic

Python ☆ 773 52

THE TALES OF CASTLE KILMERE

CHARACTERS

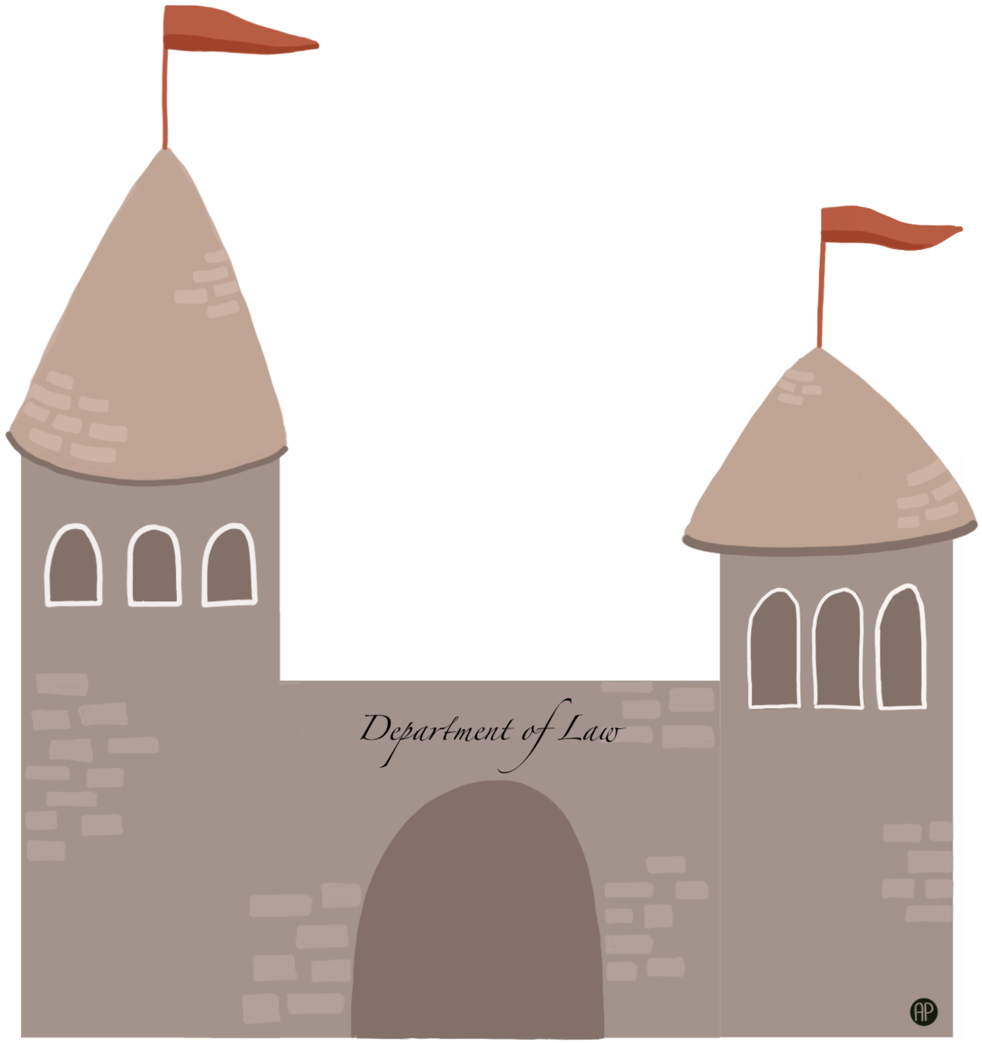
CHARACTERS



CASTLE KILMERE

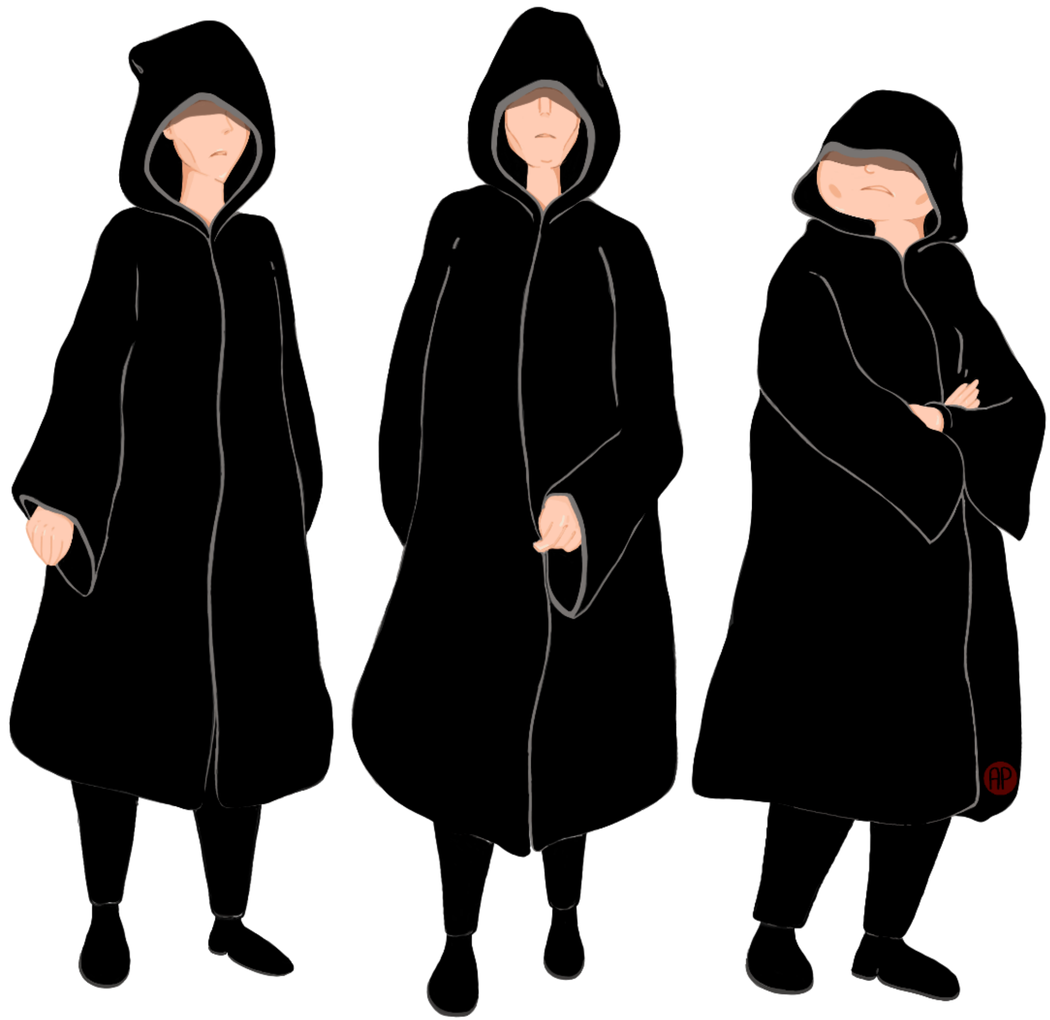
HEADMISTRESS

FACULTIES



CASTLE KILMERE

DARK ARMY



CLASSES

OBJECT ORIENTED PROGRAMMING (OOP)

- Programs contain objects that interact with each other
- Objects can contain attributes and methods
- OOP represents the structure of the real world

WHAT IS A CLASS?

- Acts as a *blueprint* for an object
- Describes how members of a class are structured, and which attributes and methods they have

CREATING A CLASS

```python
class CastleKilmereMember:
    pass

kilmere_member = CastleKilmereMember()
```

ADDING ATTRIBUTES AND METHODS

```python
class CastleKilmereMember:
    def __init__(self, name, birthyear, sex):
        self.name = name
        self.birthyear = birthyear
        self.sex = sex

    def says(self, words):
        return f"{self.name} says: {words}"
```

ADDING ATTRIBUTES AND METHODS

```
class CastleKilmereMember:
    def __init__(self, name, birthyear, sex):
        ...

miranda = CastleKilmereMember('Miranda Mirren',
                              1962,
                              'female')

print(miranda.says("Hello my dear"))
>>> Miranda Mirren says: Hello my dear
```

`__init__`

- Called when creating a new instance of a class
- The first argument of `__init__` is `self`
- `self` points towards an instance of the class

INHERITANCE

- Problem: We want more classes (pupils, professors, ghosts, …)
- All of these are members of Castle Kilmere
- This is what *inheritance* is used for
- Allows us to create a new class that inherits all attributes/methods from the parent class
- The child class can override attributes/methods of the parent class and add new functionality

INHERITANCE

```
class Pupil(CastleKilmereMember):
    def __init__(self, name, birthyear, sex, start_year, pet=N
        super().__init__(name, birthyear, sex)
        self.start_year = start_year

        if pet is not None:
            self.pet_name, self.pet_type = pet

        self._elms = {
                  'Critical Thinking': False,
                  'Self-Defense Against Fresh Fruit': False,
                  'Broomstick Flying': False,
                  'Magical Theory': False,
                  'Foreign Magical Systems': False,
```

INHERITANCE

```
lissy = Pupil(name='Lissy Spinster',
              birthyear=2008,
              sex='female',
              start_year=2020,
              pet=('Ramses', 'cat'))
```

SUMMARY

- In Python, classes implement the OOP paradigm
- Classes act as a blueprint for an object
- Parent class: `CastleKilmereMember`
- Child class: `Pupil`

TYPES OF CLASS METHODS

TYPES OF CLASS METHODS

1. Instance methods
2. Class methods
3. Static methods

INSTANCE METHODS

- Most common type of method
- At least one input parameter (`self`)
- Can modify both *object state* and *class state*

INSTANCE METHODS

Our base class already has an instance method:

```
class CastleKilmereMember:
    def __init__(self, name, birthyear, sex):
        ...

    def says(self, words):
        return f"{self.name} says {words}"
```

CLASS METHODS

- Take at least `cls` as an input
- `cls` points towards the *class*
- Can modify class state but not object state
- Applied using the `@classmethod` decorator

NOTE

The names `self` and `cls` are only conventions

ALTERNATIVE CONSTRUCTORS

- A class can only have one constructor (`__init__`)
- We can use class methods to create additional constructors

ALTERNATIVE CONSTRUCTORS

```python
class Pupil(CastleKilmereMember):
    ...

    @classmethod
    def lissy(cls):
        return cls('Lissy Spinster',
                   2008,
                  'female',
                   2018,
                  ('Ramses', 'cat'))

lissy = Pupil.lissy()
```

STATIC METHODS

- Take neither `self` nor `cls` as an input
- Cannot modify object state or class state
- Related to the class but yet independent
- Can only access data they are provided with

STATIC METHODS

```python
class Pupil(CastleKilmereMember):
    ...

    @staticmethod
    def passed(grade):
        grades = {
                'E': True,
                'Excellent': True,
                'O': True,
                'Ordinary': True,
                'A': True,
                'Acceptable': True,
                'P': False,
                'Poor': False,
```

STATIC METHODS

```
lissy = Pupil.lissy()
luke = Pupil.luke()

print(lissy.passed('O'))
>>> True
print(luke.passed('H'))
>>> False
print(Pupil.passed('O'))
>>> True
```

WHY DO WE NEED CLASS/STATIC METHODS?

- They allow developers to communicate their intention
- Example: static method expresses independence
- Class methods can be used as alternative constructors

SUMMARY

- Three types of methods: instance, class and static methods
- Note: using static methods is controversial

DEFAULTDICT

THE `collections` MODULE

- Contains several useful classes
- Especially helpful for the magical universe: `collections.defaultdict`

EXTENDING `CastleKilmereMember`

```
class CastleKilmereMember:
    def __init__(self, ...):
        ...
        self._traits = {}

    def add_trait(self, trait: str, value: bool = True):
        self._traits[trait] = value

    def print_traits(self):
        ...
```

```python
bromley = CastleKilmereMember(
          'Bromley Huckabee', 1959, 'male'
          )
bromley.add_trait("kind")
bromley.add_trait("tidy-minded")
bromley.add_trait("impatient", value=False)

bromley.print_traits()
>>> Bromley Huckabee is kind and tidy-minded
>>> Bromley Huckabee is not impatient
```

Task: add method that checks if a Castle Kilmere member exhibits a certain character trait

```
class CastleKilmereMember:
    def __init__(self, ...):
        ...
        self._traits = {}

    def exhibits_trait(self, trait: str) -> bool:
        ...
```

`CastleKilmereMember`

- `exhibits_trait` should return `True` if a trait exists and `False` if it doesn't
- Solution one: `dict.get()`
- Solution two: `collections.defaultdict`

dict.get()

```
class CastleKilmereMember:

    def __init__(self, ...):
        ...
        self._traits = {}

    def add_trait(self, trait: str, value=True):
        self._traits[trait] = value

    def exhibits_trait(self, trait: str) -> bool:
        value = self._traits.get(trait, False)
        return value
```

`collections.defaultdict`

- Subclass of the general dictionary type
- Allows us to specify a callable whose return value will be used for missing items

BASIC USAGE

```
from collections import defaultdict
my_dict = defaultdict(default_factory)
```

`collections.defaultdict`

- Requires a callable `default_factory` as an argument
- The default value is returned whenever a requested key cannot be found

`CastleKilmereMember`

- Goal: return `False` as a default value
- Maybe `dict_ = defaultdict(False)`?
- Why is this wrong?

`CastleKilmereMember`

- `defaultdict` requires a callable as an argument
- `False` is not callable
- We have to define a function that returns `False` when called without arguments

CastleKilmereMember

```python
def return_false() -> bool:
    return False

dict_ = defaultdict(return_false)
```

`CastleKilmereMember`

Alternative:

```
dict_ = defaultdict(lambda: False)
```

THE POWER OF `defaultdict`

`defaultdict` can be provided with *any* kind of callable

USE CASE - GROUPING ITEMS

```
from collections import defaultdict

pets = [('Cotton', 'owl'), ('Ramses', 'cat'),
        ('Twiggles', 'owl'), ('Oscar', 'cat'),
        ('Louie', 'cat'), ('Bob', 'ferret'),
        ('Winston', 'owl'), ('Harry', 'owl')]
```

USE CASE - GROUPING ITEMS

```
from collections import defaultdict

pets = [('Cotton', 'owl'), ('Ramses', 'cat'),
        ('Twiggles', 'owl'), ('Oscar', 'cat'),
        ('Louie', 'cat'), ('Bob', 'ferret'),
        ('Winston', 'owl'), ('Harry', 'owl')]

types_of_pets = defaultdict(list)
for name, type_ in pets:
    types_of_pets[type_].append(name)

# What is the output of the following?
for key, value in types_of_pets.items():
    print(f"{key}: {value}")
```

```
for key, value in types_of_pets.items():
    print(f"{key}: {value}")

>>> owl: ['Cotton', 'Twiggles', 'Winston', 'Harry']
>>> cat: ['Ramses', 'Oscar', 'Louie']
>>> ferret: ['Bob']
```

SUMMARY

- Common problem: accessing keys in a dictionary that don’t exist
- Can be handled with `collections.defaultict`
- Behaves nearly identical to a regular Python dictionary
- Difference: keys are created and populated with a default value when trying to access or modify a missing key

DECORATORS

WHAT ARE DECORATORS?

- Short answer: callable that takes a callable as an input and returns a callable
- Allow us to extend and/or modify the behavior of the input callable
- The decorated callable is *not* permanently modified

The simplest decorator returns its input function

```python
def useless_decorator(function):
    return function
```

We apply a decorator to a function by wrapping it

```
def say_hello():
    return f"Hey there!"

say_hello = useless_decorator(say_hello)
```

Alternative syntax

```
@useless_decorator
def say_hello():
    return f"Hey there!"
```

MODIFYING THE WRAPPED FUNCTION’S BEHAVIOR

- To modify behavior a decorator has to define a wrapper function
- This wraps the input function and modifies its behavior

EXAMPLE

```python
def goodbye(function):
    def wrapper():
        original_output = function()
        new_output = original_output + \
                     f" Goodbye, have a good day!"
        return new_output
    return wrapper

@goodbye
def say_hello():
    return f"Hey there!"

# What is the output of this print statement?
print(say_hello())
```

EXAMPLE

```python
def goodbye(function):
    def wrapper():
        original_output = function()
        new_output = original_output + 
                     f" Goodbye, have a good day!"
        return new_output
    return wrapper

@goodbye
def say_hello():
    return f"Hey there!"

print(say_hello())
>>> Hey there! Goodbye, have a good day!
```

FUNCTIONS WITH INPUT ARGUMENTS

```
def say_words(person, words):
    return f"{person} says: {words}

print(say_words("Lissy", "Hey Luke!
>>> Lissy says: Hey Luke!
```

FUNCTIONS WITH INPUT ARGUMENTS

- How can we decorate this function?
- The `goodbye` wrapper function must be able to process the inputs `person` and `words`
- Solution: use `*args` and `**kwargs`

FUNCTIONS WITH INPUT ARGUMENTS

```python
def goodbye(function):
    def wrapper(*args, **kwargs):
        original_output = function(*args, **kwargs)
        new_output = original_output +
                     f" Goodbye, have a good day!"
        return new_output
    return wrapper

@goodbye
def say_words(person, words):
    return f"{person} says: {words}"

# What is the output of this print statement?
print(say_words("Lissy", "Hey Luke!"))
```

```
print(say_words("Lissy", "Hey Luke!"))
>>> Lissy says: Hey Luke! Goodbye, have a good day!
```

WHY ARE DECORATORS CALLED DECORATORS?

Because they “decorate” other functions and allow us to run code before and after the wrapped function is executed.

SUMMARY

- Decorators allow us to modify the behaviour of a function
- The decorated function only changes when it’s decorated
- Decorators are a complex topic and we have only scratched the surface

ABSTRACT BASE CLASSES

MAGICAL UNIVERSE

- Parent class (`CastleKilmereMember`)
- Several child classes (`Pupil`, `Professor`, etc.)
- Child classes inherit all methods from their parent class
- In some cases simple inheritance is not sufficient

ABSTRACT BASE CLASSES (ABCS)

- Useful if an application involves a *hierarchy* of classes
- In this hierarchy:
  - It should be impossible to instantiate the base class
  - All subclasses should have a common base class
  - All subclasses should implement certain methods defined in the base class

EXAMPLE

EXAMPLE

```
from abc import ABC, abstractmethod

class Spell(ABC):
    def __init__(self, name, incantation, effect):
        self.name = name
        self.incantation = incantation
        self.effect = effect

    @abstractmethod
    def cast(self):
        pass

    @property
    @abstractmethod
    def defining_feature(self):
```

INTROSPECTION

```
Spell.__abstractmethods__
>>> frozenset({'cast', 'defining_feature'})
```

Can we instantiate the `Spell` class?

```
stuporus = Spell(name='The stuporus ratiato spell',
                 incantation='Stuporus Ratiato',
                 effect='Makes objects fly')
```

```python
stuporus = Spell(name='The stuporus ratiato spell',
                 incantation='Stuporus Ratiato',
                 effect='Makes objects fly')

>>> TypeError: Can't instantiate abstract class Spell with
    abstract methods cast, defining_feature
```

SUBCLASS `Charm`

```
class Charm(Spell):
    def __init__(self, name, incantation,
                 effect, difficulty, min_year):
        super().__init__(name, incantation, effect)
        self.difficulty = difficulty
        self.min_year = min_year

    def cast(self):
        print(f"{self.incantation}!")
```

Can we instantiate the `Charm` class?

```
stuporus = Charm(name="The stuporus ratiato spell",
                 incantation="Stuporus Ratiato",
                 effect="Makes objects fly",
                 difficulty="Simple")
```

```
stuporus = Charm(name="The stuporus ratiato spell",
                 incantation="Stuporus Ratiato",
                 effect="Makes objects fly",
                 difficulty="Simple")

>>> TypeError: Can't instantiate abstract class Charm
    with abstract methods defining_feature
```

But why?

ANSWER

Because we forgot to implement the `defining_feature` method!

ABC VS. NORMAL CLASS

- Situation: a subclass does not implement all methods required by the base class
- When does a normal class raise an error?
- Answer: only when calling the missing method
- When does an ABC raise an error?
- Answer: at *instantiation time*

```python
class Charm(Spell):
    def __init__(self, name, incantation,
                 effect, difficulty, min_year):
        super().__init__(name, incantation, effect)
        self.difficulty = difficulty
        self.min_year = min_year

    @property
    def defining_feature(self):
        return ("Alteration of the object's inherent
                 qualities, that is, its behaviour
                 and capabilities")

    def cast(self):
        print(f"{self.incantation}!")
```

```
stuporus = Charm(name="The stuporus ratiato spell",
                 incantation="Stuporus Ratiato",
                 effect="Makes objects fly",
                 difficulty="Simple")

print(stuporus)
>>> Charm(name="The stuporus ratiato spell",
          incantation='Stuporus Ratiato',
          effect='Makes objects fly',
          difficulty='Simple')
```

SUMMARY

- ABCs formalize the relationship between a parent class and a subclass
- Three purposes:
  - Allow the parent class to demand a certain structure of their subclasses
  - Allow subclasses to identify as meeting those requirements
  - Enforce that a subclass meets the requirements
- ABCs are a huge topic - we have only seen a small part of it

NAMEDTUPLES

What is a tuple?

TUPLES

- Data structure for grouping arbitrary objects
- Tuples are *immutable*

TUPLES

The `pet` attribute of the `Pupil` class is a tuple:

```
pet = ('name', 'type')
lissys_pet = ('Ramses', 'cat')
```

TUPLES

```
lissys_pet = ('Ramses', 'cat')
print(lissys_pet[0])
>>> Ramses
```

TUPLES ARE IMMUTABLE

```
lissys_pet[0] = 'Twiggles'
>>> TypeError: 'tuple' object does not support
    item assignment
```

NAMEDTUPLES

- Variation of plain tuples
- Allow us to name the fields of the tuple
- Easier to access individual fields
- Makes code more readable

CREATING NAMEDTUPLES

- `collections.namedtuple`
- `typing.NamedTuple`

`typing.NamedTuple`

- Allows us to specify the type of each field
- Makes it easy to add methods to the class

typing.NamedTuple

```
from typing import NamedTuple

class Pet(NamedTuple):
    name: str
    type: str

lissys_pet = Pet('Ramses', 'cat')

print(lissys_pet)
>>> Pet(name='Ramses', type='cat')
```

ACCESSING FIELDS

```
name = lissys_pet[0]
# or
name = lissys_pet.name
```

MAGICAL UNIVERSE

- Pupils, professors and ghosts should not be immutable
- Suitable group of people: Dark Army members

DARK ARMY

DARK ARMY CLASS

```python
from typing import NamedTuple

class DarkArmyMember(NamedTuple):
    name: str
    birthyear: int

    @classmethod
    def leader(cls) -> 'DarkArmyMember':
        return cls('Master Odon', 1971)
```

DARK ARMY CLASS

From now on we can easily create new Dark Army members:

```
keres = DarkArmyMember('Keres Fulford', 1983)

print(keres)
>>> DarkArmyMember(name='Keres Fulford', birthyear=1983)

print(keres.leader())
>>> DarkArmyMember(name='Master Odon', birthyear=1971)

keres.name = "Mortimer"
>>> AttributeError: can't set attribute
```

SUMMARY

- Namedtuples are an extension of plain tuples
- They present a shortcut for creating immutable classes
- With Python 3.7 we can also use data classes for creating immutable classes

THE END