Package Structure and Text Processing

Matthew Seguin

Preliminary Importing of Packages

```
import re
import statsmodels
import pandas as pd
from plotnine import *
```

1.

a.

We can import stats models then use dir to check the contents of the namespace created.

```
dir(statsmodels)
```

```
['__all__', '__builtins__', '__cached__', '__doc__', '__file__', '__loader__', '__name__', '__package__', '__path__', '__spec__', '__version__', '__version_info__', '__version_tuple__', '_version', 'compat', 'debug_warnings', 'monkey_patch_cat_dtype', 'test', 'tools']
```

```
for i in dir(statsmodels):
  spaces = " :"
  for j in range(22 - len(i)):
    spaces += " "
  print(i, spaces, type(getattr(statsmodels,i)))
```

```
__all__  :               <class 'list'>
__builtins__  :          <class 'dict'>
__cached__  :            <class 'str'>
__doc__  :               <class 'NoneType'>
__file__  :              <class 'str'>
__loader__  :            <class '_frozen_importlib_external.SourceFileLoader'>
__name__  :              <class 'str'>
__package__  :           <class 'str'>
__path__  :              <class 'list'>
__spec__  :              <class '_frozen_importlib.ModuleSpec'>
__version__  :           <class 'str'>
__version_info__  :      <class 'tuple'>
__version_tuple__  :     <class 'tuple'>
_version  :              <class 'module'>
compat  :                <class 'module'>
debug_warnings  :        <class 'bool'>
monkey_patch_cat_dtype  : <class 'function'>
test  :                  <class 'function'>
tools  :                 <class 'module'>
```

As we can see there are a number of double underscore objects ranging from “all” to “version_tuple”, then there are a number of submodules. The types of all objects in the name space are displayed.

First to get the absolute file path to the statsmodels package we can use the dunder “file” object.

```
init_path = statsmodels.__file__
package_path = init_path[:-12]
package_path
```

```
'/usr/.../.local/lib/python3.11/site-packages/statsmodels'
```

Now we can see one of the submodules is called _version so this is a reasonable place to start looking for the version of our statsmodels package.

```
dir(statsmodels._version)
```

```
['__builtins__', '__cached__', '__doc__', '__file__', '__loader__', '__name__', '__package__', '__spec__', '__version__', '__version_tuple__', 'version', 'version_tuple']
```

```
statsmodels._version.__version__
```

```
'0.14.0'
```

First we see that the _version submodule has a double underscore "version" object. So I check the file contents of _version.py

```
statmodeldir="/…/statsmodels"
cat "/…/statsmodels/_version.py"
```

```
# coding: utf-8
# file generated by setuptools_scm
# don't change, don't track in version control
__version__ = version = '0.14.1'
__version_tuple__ = version_tuple = (0, 14, 1)
```

As we can see we already have our answer, the version number is in the _version submodule file.

b.

```
statmodeldir="/…/statsmodels"
cat "$statmodeldir/api.py"
```

```
# -*- coding: utf-8 -*-

__all__ = [
    "BayesGaussMI",
    "BinomialBayesMixedGLM",
    "ConditionalLogit",
    "ConditionalMNLogit",
    "ConditionalPoisson",
    "Factor",
    "GEE",
    "GLM",
    "GLMGam",
    "GLS",
    "GLSAR",
    "GeneralizedPoisson",
    "HurdleCountModel",
    "Logit",
    "MANOVA",
    "MI",
    "MICE",
    "MICEData",
    "MNLogit",
    "MixedLM",
    "NegativeBinomial",
    "NegativeBinomialP",
    "NominalGEE",
    "OLS",
    "OrdinalGEE",
    "PCA",
    "PHReg",
    "Poisson",
    "PoissonBayesMixedGLM",
    "ProbPlot",
    "Probit",
    "QuantReg",
    "RLM",
    "RecursiveLS",
    "SurvfuncRight",
    "TruncatedLFPoisson",
    "TruncatedLFNegativeBinomialP",
    "WLS",
    "ZeroInflatedGeneralizedPoisson",
    "ZeroInflatedNegativeBinomialP",
    "ZeroInflatedPoisson",
    "__version__",
```

```
    "add_constant",
    "categorical",
    "cov_struct",
    "datasets",
    "distributions",
    "duration",
    "emplike",
    "families",
    "formula",
    "gam",
    "genmod",
    "graphics",
    "iolib",
    "load",
    "load_pickle",
    "multivariate",
    "nonparametric",
    "qqline",
    "qqplot",
    "qqplot_2samples",
    "regression",
    "robust",
    "show_versions",
    "stats",
    "test",
    "tools",
    "tsa",
    "webdoc",
    "__version_info__"
]


from . import datasets, distributions, iolib, regression, robust, tools
from .__init__ import test
from statsmodels._version import (
    version as __version__, version_tuple as __version_info__
)
from .discrete.conditional_models import (
    ConditionalLogit,
    ConditionalMNLogit,
    ConditionalPoisson,
)
from .discrete.count_model import (
    ZeroInflatedGeneralizedPoisson,
    ZeroInflatedNegativeBinomialP,
    ZeroInflatedPoisson,
)
from .discrete.discrete_model import (
    GeneralizedPoisson,
    Logit,
    MNLogit,
    NegativeBinomial,
```

```
    NegativeBinomialP,
    Poisson,
    Probit,
)
from .discrete.truncated_model import (
    TruncatedLFPoisson,
    TruncatedLFNegativeBinomialP,
    HurdleCountModel,
    )
from .duration import api as duration
from .duration.hazard_regression import PHReg
from .duration.survfunc import SurvfuncRight
from .emplike import api as emplike
from .formula import api as formula
from .gam import api as gam
from .gam.generalized_additive_model import GLMGam
from .genmod import api as genmod
from .genmod.api import (
    GEE,
    GLM,
    BinomialBayesMixedGLM,
    NominalGEE,
    OrdinalGEE,
    PoissonBayesMixedGLM,
    cov_struct,
    families,
)
from .graphics import api as graphics
from .graphics.gofplots import ProbPlot, qqline, qqplot, qqplot_2samples
from .imputation.bayes_mi import MI, BayesGaussMI
from .imputation.mice import MICE, MICEData
from .iolib.smpickle import load_pickle
from .multivariate import api as multivariate
from .multivariate.factor import Factor
from .multivariate.manova import MANOVA
from .multivariate.pca import PCA
from .nonparametric import api as nonparametric
from .regression.linear_model import GLS, GLSAR, OLS, WLS
from .regression.mixed_linear_model import MixedLM
from .regression.quantile_regression import QuantReg
from .regression.recursive_ls import RecursiveLS
from .robust.robust_linear_model import RLM
from .stats import api as stats
from .tools.print_version import show_versions
from .tools.tools import add_constant, categorical
from .tools.web import webdoc
from .tsa import api as tsa

load = load_pickle
```

First the datasets, distributions, iolib, regression, robust, and tools submodules are all imported so each of the double underscore "init" files for these modules is called. Then the main "init" file in the directory of

statsmodels is accessed in order to import the test function contained within it. Then the _version.py file which we saw in the previous part is accessed in order to bring in the version information. Then from each of the submodules conditional_models.py, count_model.py, discrete_model.py, and truncated_model.py from the discrete subpackage a number of classes are imported. Then each of the submodules api, hazard_regression, and survfunc from the duration subpackage are accessed. Then the emplike, formula, gam, and genmod subpackages are accessed where the api.py file is accessed for each and for gam and genmod other submodules are also accessed. Finally a number of files in each of the subpackages graphics, imputation, iolib, multivariate, nonparametric, regression, robust, stats, tools, and tsa are accessed.

It seems like the developers of this package decided to make an api.py module for any larger subpackage. When accessing all of these submodules a number of other submodules are also accessed along with completely different packages (ex: numpy), it would be too much to go over every file accessed.

We can see that the MICE class was imported from imputation.mice as shown from the api before.

```
statmodeldir="/…/statsmodels"
cat "$statmodeldir/api.py" | grep "MICE"
echo # just adding space
echo # just adding space
cat "$statmodeldir/imputation/mice.py" | grep -A 7 "class MICE"
```

```
    "MICE",
    "MICEData",
from .imputation.mice import MICE, MICEData


class MICEData:

    __doc__ = """\
    Wrap a data set to allow missing data handling with MICE.

    Parameters
    ----------
    data : Pandas data frame
--
class MICE:

    __doc__ = """\
    Multiple Imputation with Chained Equations.

    This class can be used to fit most statsmodels models to data sets
    with missing values using the 'multiple imputation with chained
    equations' (MICE) approach..
--
class MICEResults(LikelihoodModelResults):

    def __init__(self, model, params, normalized_cov_params):

        super(MICEResults, self).__init__(model, params,
                                          normalized_cov_params)

    def summary(self, title=None, alpha=.05):
```

We can see there is a mice.py which is where MICE was being imported from. By searching mice.py for MICE we can see that MICE is a class object imported from the imputation subpackage in the mice.py module. MICE is a class used to fit most statsmodels models to data sets with missing values using the 'multiple imputation with chained equations' (MICE) approach.

In a similar fashion I will search for the GLM class.

```
statmodeldir="/…/statsmodels"
cat "$statmodeldir/api.py" | grep -B 2 "GLM"
echo # just adding space
echo # just adding space
cat "$statmodeldir/genmod/api.py" | grep "GLM"
echo # just adding space
echo # just adding space
cat "$statmodeldir/genmod/generalized_linear_model.py" | grep "class GLM"
echo # just adding space
echo # just adding space
cat "$statmodeldir/genmod/generalized_linear_model.py" | grep -E "import.+base"
```

```
__all__ = [
    "BayesGaussMI",
    "BinomialBayesMixedGLM",
--
    "Factor",
    "GEE",
    "GLM",
    "GLMGam",
--
    "PHReg",
    "Poisson",
    "PoissonBayesMixedGLM",
--
from .formula import api as formula
from .gam import api as gam
from .gam.generalized_additive_model import GLMGam
--
from .genmod.api import (
    GEE,
    GLM,
    BinomialBayesMixedGLM,
    NominalGEE,
    OrdinalGEE,
    PoissonBayesMixedGLM,

    "GLM", "GEE", "OrdinalGEE", "NominalGEE",
    "BinomialBayesMixedGLM", "PoissonBayesMixedGLM",
from .generalized_linear_model import GLM
from .bayes_mixed_glm import BinomialBayesMixedGLM, PoissonBayesMixedGLM


class GLM(base.LikelihoodModel):
class GLMResults(base.LikelihoodModelResults):
```

```
class GLMResultsWrapper(lm.RegressionResultsWrapper):


import statsmodels.base.model as base
import statsmodels.base.wrapper as wrap
import statsmodels.base._parameter_inference as pinfer
```

We can see there is an api.py which is where GLM was being imported from. From searching api.py for GLM we see it is being imported from the generalized_linear_model submodule (which we saw the .py file for in the genmod folder before) so next I search there. From searching there we see GLM is a class object that inherits from the LikelihoodModel class in the base module and I search for how base is being imported next.

As one might expect it comes from the base subpackage of statsmodels, and in particular the LikelihoodModel class comes from the model module in that subpackage. Which by the following we can see inherits from the Model base class.

```
statmodeldir="/…/statsmodels"
cat "$statmodeldir/base/model.py" | grep "class LikelihoodModel"
echo # just adding space
echo # just adding space
cat "$statmodeldir/base/model.py" | grep "class Model"
```

```
class LikelihoodModel(Model):
class LikelihoodModelResults(Results):


class Model:
```

Therefore we have our final answer that GLM is a class object that inherits from the LikelihoodModel class which inherits from the Model base class (both of which come from the model module of the base subpackage), and GLM is imported from api.py in the genmod subpackage which first imports it from generalized_linear_model.py in the genmod subpackage.

We can check this with python:

```
import statsmodels.api as sm
type(sm.MICE)
```

```
<class 'type'>
```

```
type(sm.GLM)
```

```
<class 'type'>
```

c.

First we will just check in python what is in the namespace.

```
dir(sm.gam)
```

```
['BSplines', 'CyclicCubicSplines', 'GLMGam', 'MultivariateGAMCVPath', '__all__', '__builtins__', '__cached__', '__doc__', '__file__', '__loader__', '__name__', '__package__', '__spec__']
```

In order to examine how the importing works we will first search the original api.py file for gam.

```
statmodeldir="/.../statsmodels" cat "$statmodeldir/api.py" | grep "gam"
echo # just adding space
echo # just adding space
cat "$statmodeldir/gam/api.py"
```

```
    "gam",
from .gam import api as gam
from .gam.generalized_additive_model import GLMGam


from .generalized_additive_model import GLMGam
from .gam_cross_validation.gam_cross_validation import MultivariateGAMCVPath
from .smooth_basis import BSplines, CyclicCubicSplines

__all__ = ["BSplines", "CyclicCubicSplines", "GLMGam", "MultivariateGAMCVPath"]
```

We can see the gam subpackage is imported via its api so first I examine that and see that gam imports all of the classes we saw in the namespace before from a number of other modules (which we will explore below).

First generalized_additive_model is accessed to get GLMGam so now we will look at that.

```
statmodeldir="/.../statsmodels"
cat "$statmodeldir/gam/generalized_additive_model.py" | grep "GLMGam"
```

```
class GLMGamResults(GLMResults):
    GLMGamResults inherits from GLMResults
        super(GLMGamResults, self).__init__(model, params,
        predict_results = super(GLMGamResults, self).predict(ex,
        return super(GLMGamResults, self).get_prediction(ex, transform=False,
class GLMGamResultsWrapper(GLMResultsWrapper):
wrap.populate_wrapper(GLMGamResultsWrapper, GLMGamResults)
class GLMGam(PenalizedMixin, GLM):
    _results_class = GLMGamResults
    _results_class_wrapper = GLMGamResultsWrapper
        super(GLMGam, self).__init__(endog, exog=exog, family=family,
        """estimate parameters and create instance of GLMGamResults class
        res : instance of wrapped GLMGamResults
            res = super(GLMGam, self).fit(start_params=start_params,
        glm_results = GLMGamResults(self, wls_results.params,
        return GLMGamResultsWrapper(glm_results)
                                    gam=GLMGam, cost=cost, endog=self.endog,
```

So we see GLMGam in the sm.gam namespace comes from generalized_additive_model.py

Now we will look in gam_cross_validation.gam_cross_validation as that is where MultivariateGAMCVPath is imported from.

```
statmodeldir="/.../statsmodels"
cat "$statmodeldir/gam/gam_cross_validation/gam_cross_validation.py" | grep "MultivariateGAMCVPath"
```

```
class MultivariateGAMCVPath:
```

So we see MultivariateGAMCVPath in the sm.gam namespace comes from gam_cross_validation.py

Now we will look in smooth_basis as that is where BSplines and CyclicCubicSplines are imported from.

```
statmodeldir="/.../statsmodels"
cat "$statmodeldir/gam/smooth_basis.py" | grep "BSplines"
echo # just adding space
echo # just adding space
cat "$statmodeldir/gam/smooth_basis.py" | grep "CyclicCubicSplines"
echo # just adding space
echo # just adding space
cat "$statmodeldir/gam/smooth_basis.py" | grep "AdditiveGamSmoother"
```

```
class UnivariateBSplines(UnivariateGamSmoother):
        super(UnivariateBSplines, self).__init__(
class BSplines(AdditiveGamSmoother):
        super(BSplines, self).__init__(x, include_intercept=include_intercept,
            uv_smoother = UnivariateBSplines(


class CyclicCubicSplines(AdditiveGamSmoother):
        super(CyclicCubicSplines, self).__init__(x,


class AdditiveGamSmoother(with_metaclass(ABCMeta)):
class GenericSmoothers(AdditiveGamSmoother):
class PolynomialSmoother(AdditiveGamSmoother):
class BSplines(AdditiveGamSmoother):
class CubicSplines(AdditiveGamSmoother):
class CyclicCubicSplines(AdditiveGamSmoother):
```

BSplines inherits from AdditiveGamSmoother so I search for that and see that AdditiveGamSmoother inherits from with_metaclass(ABCMeta) so let us search for with_metaclass

```
statmodeldir="/.../statsmodels"
cat "$statmodeldir/gam/smooth_basis.py" | grep "with_metaclass"
echo # just adding space
echo # just adding space
cat "$statmodeldir/compat/python.py" | grep "with_metaclass"
```

```
from statsmodels.compat.python import with_metaclass
class UnivariateGamSmoother(with_metaclass(ABCMeta)):
class AdditiveGamSmoother(with_metaclass(ABCMeta)):

    "with_metaclass",
def with_metaclass(meta, *bases):
```

First we see with_metaclass comes from compat.python so I search there and see that it is a function that returns a class. ABCMeta comes from another package but the details of this class aren't that important.

So we see BSplines and CyclicCubicSplines come from smooth_basis.py and inherit from AdditiveGamSmoother (which uses with_metaclass, a function not a class).

d.

First we want to look for the distributions subpackage in api.py since we know monotone_fn_inverter comes from there.

```
statmodeldir="/…/statsmodels"
cat "$statmodeldir/api.py" | grep "distributions"
echo # just adding space
echo # just adding space
cat "$statmodeldir/distributions/__init__.py" | grep -B 1 -A 1 "monotone_fn_inverter"
```

```
    "distributions",
from . import datasets, distributions, iolib, regression, robust, tools


from .empirical_distribution import (
    ECDF, ECDFDiscrete, monotone_fn_inverter, StepFunction
    )
--
    'genpoisson_p',
    'monotone_fn_inverter',
    'test',
```

We see distributions is simply imported (directly from its init file) so I look in there and see that monotone_fn_inverter is imported from the empirical_distribution module. Now I will look in there.

```
statmodeldir="/…/statsmodels"
cat "$statmodeldir/distributions/empirical_distribution.py" | grep -A 17 "monotone_fn_inverter"
```

```
def monotone_fn_inverter(fn, x, vectorized=True, **keywords):
    """
    Given a monotone function fn (no checking is done to verify monotonicity)
    and a set of x values, return an linearly interpolated approximation
    to its inverse from its values on x.
    """
    x = np.asarray(x)
    if vectorized:
        y = fn(x, **keywords)
    else:
        y = []
        for _x in x:
            y.append(fn(_x, **keywords))
        y = np.array(y)

    a = np.argsort(y)

    return interp1d(y[a], x[a])
```

We see that monotone_fn_inverter is a function from empirical_distribution.py that is imported into the distributions subpackage and imported into the main package from there. The function inverts a monotone function using interpolation after given a set of x values.

2.

First we start by importing the data using the function given. I use one class (designed off of the output of the given file to get the data) to do this entire question, here is a little about it's fields and methods:

ChunkedDebate Takes In:

- string: A string object of debate text where the indication of a new chunk is given by "SPEAKER:" with possibly a space after
- candidates: A list of dictionaries containing the candidates (Dem and Rep being the keys)
- moderators: A list of moderators (which seems to just be speakers)

ChunkedDebate Has Attributes:

- string: Just the raw debate string passed to it
- candidates: A list of candidates for the given debate
- moderators: A list of moderators for the given debate
- year: The year the debate occured in
- cans: A list of all candidates over any set of debates
- speakers: A list of all speakers in any set of debates
- all_regex: A regex that will match any "SPEAKER:" with a possible space after
- starts,ends,groups: lists of match information in string from using all_regex
- chunk_locations: A dictionary containing the speaker, start, and end position information of each chunk
- Deb_Info: A string containing the information portion before anyone speaks in string
- matches: A pandas dataframe containing the speaker, text, and order spoken in debate information for each chunk
- chunk_counts: A pandas dataframe that contains the number of chunks for each speaker
- words: A dictionary containing all of the words spoken by each speaker
- words_stats: A pandas dataframe containing the number of words, characters, and average number of characters per word for each candidate in each debate

ChunkedDebate Has Method:

- init: Construction method for calling all other methods
- get_candidates_list: Makes a list of all the candidates (they were in a dictionary before)
- get_speakers: Makes a list of all the speakers (moderators and candidates)
- speaker_regex: Construct all_regex that will match any "SPEAKER:" with a possible space after
- get_year: Gets the year the debate was held in
- location_finder: Gets match information for searching through string with all_regex
- combine_same_speaker: Combines adjacent chunks of the same speaker for the location information
- text_exctractor: Uses chunk location information to exctract chunk text, speaker, and order spoken and hold in pandas dataframe

- IdentifyModerator: Determines if the moderator is announced at the start by MODERATOR: and adjusts chunk data and debate info as needed
- speaker_counts: Gets the number of chunks per speaker
- get_words: Gets all of the individual words spoken by each speaker
- get_word_stats: Get the number of words/characters and avg characters for each candidate

I combine all of the candidates and moderators into a vector representing all of the speakers. Then I take only the unique speakers.

We see that in the text the indication of who is speaking is given by SPEAKER: so I make a regex that will match any of these based on the speakers we have. Then I get all of the matches, starting positions, and ending positions from this regex in the text. Then if there are any speakers next to each other that are the same in the list we combine those into one, taking the earliest start position (the first one) and the later end position (the second one) and return a dictionary containing this information. I also get the year of each debate.

Then based on these locations I take the sections of text from start to end for each of these, getting rid of non spoken text and the SPEAKER: with regex (and I get the debate info which is what comes before the first speaker match). I remove all formatting from these chunks add them to a dataframe. Then I noticed that sometimes at the beginning the moderator is specified in the text for example by MODERATOR:whoever it is. So I check to see if there is only one occurence of MODERATOR and if there is I move that data from the dataframe into the debate info. I get the number of chunks for each speaker from our dataframe.

Then I get the individual words spoken by each speaker and some statistics about those words by speaker.

```
with open("prob3.py") as f:
  script = f.read()
  exec(script)
```

```
October 3, 2000 Transcript
October 13, 2004 Debate Transcript
October 15, 2008 Debate Transcript
October 3, 2012 Debate Transcript
September 26, 2016 Debate Transcript
September 29, 2020 Debate Transcript
October 3, 2000 TranscriptOctober 3, 2000The First Gore-Bush Presidential DebateMODERATOR: Good even
September 29, 2020 Debate TranscriptPresidential Debate at Case Western Reserve University and Cleve
```

```
class ChunkedDebate:
  def __init__(self, string, candidates = candidates, moderators = moderators):
    if type(string) is not str:
      raise TypeError("Debate Body Must be a string")
    if type(candidates) is not list:
      raise TypeError("Candidates must be a list of dictionaries with keys 'Dem' and 'Rep'")
    if len(candidates) == 0:
      raise ValueError("Candidates must have positive length")
    if type(candidates[0]) is not dict:
      raise TypeError("Candidates must be a list of dictionaries with keys 'Dem' and 'Rep'")
    if list(candidates[0].keys()) != ["Dem", "Rep"]:
      raise ValueError("The entries of candidates must have keys 'Dem' and 'Rep' only")
    if type(moderators) is not list:
      raise TypeError("Moderators must be a list")
```

```
    # Assign the raw string to our object
    self.string = string.strip()
    # Make a list of moderators
    self.moderators = moderators
    # Get the candidates
    # (note this is over all debates and will be reassigned later)
    self.candidates = candidates
    # Get all candidates in list form
    self.__get_candidate_list__()
    # Get a list of all speakers
    self.__get_speakers__()
    # Get the regex for all speakers
    self.speaker_regex()
    # Get the year of the debate
    try:
      self.__get_year__()
    except AttributeError:
      raise ValueError("Debate must contain a year")
    # Get the locations of new chunks
    self.__location_finder__()
    # Combine adjacent chunks by the same speaker
    self.__combine_same_speaker__()
    # Get the text from chunks
    self.__text_extractor__()
    # Check if the moderator is defined in the start
    self.__IdentifyModerator__()
    # Get the number of chunks by speaker
    self.__speaker_counts__()
    # Get words for each speaker
    self.__get_words__()
    # Get word statistics
    self.__get_word_stats__()

  def __get_candidate_list__(self):
    # Candidates was in a dictionary form so make a list
    cans = []
    for i in self.candidates:
      cans += [str(i["Dem"])]
      cans += [str(i["Rep"])]
    self.cans = cans
    return(cans)

  def __get_speakers__(self):
    '''
    Make a list of all the speakers given
    all moderators and candidates
    '''
    speakers = moderators
    speakers += self.cans
    # Take only unique speakers
    self.speakers = list(set(speakers))
```

```
    return(speakers)

  def speaker_regex(self):
    '''
    Takes in the list of speakers and makes a regex
    that will match any "SPEAKER:" with possibly a space
    after for any of those speakers
    '''
    speaker_match = "|".join([f"{i}:\\s?" for i in self.speakers])
    self.all_regex = speaker_match
    return(speaker_match)

  def __get_year__(self):
    '''
    Gets the year of the debate from the raw string
    '''
    self.year = int(re.search("[0-9]{4}", self.string).group())

  def __location_finder__(self):
    '''
    Gets the matches, starting positions, and ending
    positions for each match of our regex
    '''
    # Make a list of all the starting positions
    self.starts = [x.start() for x in re.finditer(self.all_regex, self.string)]
    # Same for ending positions
    self.ends = [x.end() for x in re.finditer(self.all_regex, self.string)]
    # Same for matches
    self.groups = [x.group() for x in re.finditer(self.all_regex, self.string)]
    return(None)

  def __combine_same_speaker__(self):
    '''
    Takes in all of the starting and ending positions and
    matches from our regex and combines adjacent chunks of
    the same speaker
    '''
    i = 0
    while i < len(self.groups) - 1:
      # If the next speaker is the same
      if self.groups[i] == self.groups[i+1]:
        # Keep only the first start position
        self.starts.pop(i+1)
        # Keep only the later end position
        self.ends.pop(i)
        # Remove one of the matches (they are the same)
        self.groups.pop(i)
        # We need to decrease the index to check
        # the current one with the new next one
        i -= 1
      i += 1
```

```
    chunk_locations = {"Starts": self.starts, "Ends": self.ends, "Speakers": self.groups}
    self.chunk_locations = chunk_locations
    return(chunk_locations)

  def __text_extractor__(self):
    '''
    Takes in all of the starting and ending positions and
    matches before and gets the text from each chunk, removing
    unwanted data (like the non spoken text). Also get the
    debate info at the start of the string
    '''
    matches = {"Speaker": [], "Text": [], "Order": []}
    # Get debate info
    Deb_Info_end = self.chunk_locations["Starts"][0]
    self.Deb_Info = self.string[:Deb_Info_end]
    n = len(self.chunk_locations["Speakers"])

    for i in range(n):
      # Add the current speaker
      matches["Speaker"].append(self.chunk_locations["Speakers"][i])
      # Get the start position
      start = self.chunk_locations["Starts"][i]
      # If we are at the end we want to take the length of the string
      if i == n -1:
        end = len(self.string)
      # Otherwise we want to stop just before the next chunk
      else:
        end = self.chunk_locations["Starts"][i+1]
      # Get the chunk from the string and remove bits like
      # (APPLAUSE) and [APPLAUSE]
      text = re.sub("\\([a-zA-Z]*\\)", " ", self.string[start:end])
      text = re.sub("\\[[a-zA-Z]*\\]", " ", text)
      # Remove the speaker
      text = re.sub(self.all_regex, " ", text)
      # Change and multiple spaces to single ones
      text = re.sub("\\s+", " ", text)
      matches["Text"].append(text)
      # Add the order the chunk was spoken in the debate
      matches["Order"].append(i+1)

    self.matches = pd.DataFrame(matches)
    return(matches)

  def __IdentifyModerator__(self):
    '''
    Identifies if the moderator was specified at the start
    with MODERATOR:whoever it is and removes it from our
    dataframe but adds it to the debate info
    '''
    # Get the data for MODERATOR "spoken" text
    MOD_ONLY = self.matches[self.matches["Speaker"] == "MODERATOR:"]
```

```
    # If there is only one then it was just the moderator being specified
    if len(MOD_ONLY) == 1:
      # Add the moderator name to the debate info
      MODNAME = MOD_ONLY["Text"][0]
      self.Deb_Info += f" MODERATOR: {MODNAME}"
      # Remove the moderator from our data and reinitialize
      # the indices of our data and spoken order
      self.matches = self.matches[self.matches["Speaker"] != "MODERATOR:"]
      self.matches["Order"] = self.matches.index
      self.matches.index = [x-1 for x in self.matches.index]
    return(None)

  def __speaker_counts__(self):
    '''
    Gets the number of chunks for each speaker
    '''
    self.chunk_counts = self.matches["Speaker"].value_counts()

  def __get_words__(self):
    '''
    Extracts the individual words by speaker for the debate
    '''
    # Get the speakers and make a dictionary
    speakers = [re.sub(":\\s?", "", s) for s in list(set(self.matches["Speaker"]))]
    blanks = [[] for _ in speakers]
    words_by_speaker = dict(zip(speakers, blanks))

    for i in self.matches.index:
      # Remove any unwanted text and change multiple spaces to single
      text = re.sub("[\\.\\?!;:_]", " ", self.matches["Text"][i])
      text = re.sub(",", "", text)
      text = re.sub("\\s+", " ", text)
      # Split the string to individual words
      words = text.split()
      # Remove single character "words" that aren't alphanumeric
      words = [w for w in words if len(w) > 1 or w.isalnum()]
      speaker = self.matches["Speaker"][i]
      speaker = re.sub(":\\s?", "", speaker)
      # Add the new words to our dictionary
      words_by_speaker[speaker] += words
    self.words = words_by_speaker
    return(words_by_speaker)

  def __get_word_stats__(self):
    '''
    Get the desired statistics of spoken words
    (Number of words/characters and avg characters)
    '''
    # Get the candidates for a given debate
    cans = [x for x in list(self.words.keys()) if x in self.cans]
    # Reassign candidates
```

```
    self.candidates = cans
    # Get the number of words for each candidate
    num_words = [len(self.words[x]) for x in cans]
    num_chars = []
    # Get the number of characters for each candidate
    for x in cans:
      num_chars.append(sum([len(w) for w in self.words[x]]))
    # Get the average characters per word for each candidate
    avg_chars = [float(c)/float(w) for c,w in zip(num_chars,num_words)]
    year = [self.year for _ in num_words]
    self.words_stats = pd.DataFrame({"Candidate": cans,
                                     "Num Words": num_words,
                                     "Num Chars": num_chars,
                                     "Avg Chars": avg_chars,
                                     "Year": year})
    return(None)

# Create our custom class object for each debate
debates = [ChunkedDebate(x) for x in debates_body]
```

a.

Our class was defined before now we are just showing for each debate some previews of our chunked data now and the number of chunks for each speaker.

```
for i in debates:
  print(f"Preview of {i.year} {i.candidates} debate data frame:")
  print(i.matches.head(n=3))
  print("")
  print("Number of chunks by speaker:")
  print(i.chunk_counts)
  print("\n")
```

```
Preview of 2000 ['GORE', 'BUSH'] debate data frame:
       Speaker                                               Text  Order
0  MODERATOR:    Good evening from the Clark Athletic Center a...      1
1       GORE:    Well, Jim, first of all, I would like to than...      2
2  MODERATOR:                  Governor Bush, one minute rebuttal.      3

Number of chunks by speaker:
Speaker
MODERATOR:     60
BUSH:          56
GORE:          49
Name: count, dtype: int64


Preview of 2004 ['KERRY', 'BUSH'] debate data frame:
       Speaker                                               Text  Order
0  SCHIEFFER:    Good evening from Arizona State University in...      1
1      KERRY:    Well, first of all, Bob, thank you for modera...      2
2  SCHIEFFER:                  Mr. President, you have 90 seconds.      3

Number of chunks by speaker:
Speaker
SCHIEFFER:     57
KERRY:         31
BUSH:          29
Name: count, dtype: int64


Preview of 2008 ['MCCAIN', 'OBAMA'] debate data frame:
       Speaker                                               Text  Order
0  SCHIEFFER:    Good evening. And welcome to the third and la...      1
1     MCCAIN:    Well, let - let me say, Bob, thank you.And th...      2
2  SCHIEFFER:                           All right. Senator Obama?      3

Number of chunks by speaker:
Speaker
MCCAIN:        60
SCHIEFFER:     55
```

```
OBAMA:         45
Name: count, dtype: int64


Preview of 2012 ['OBAMA', 'ROMNEY'] debate data frame:
    Speaker                                                Text  Order
0  LEHRER:    Good evening from the Magness Arena at the Un...      1
1   OBAMA:    Well, thank you very much, Jim, for this oppo...      2
2  LEHRER:                        Governor Romney, two minutes.      3

Number of chunks by speaker:
Speaker
LEHRER:     76
ROMNEY:     54
OBAMA:      42
Name: count, dtype: int64


Preview of 2016 ['TRUMP', 'CLINTON'] debate data frame:
     Speaker                                                Text  Order
0     HOLT:    Good evening from Hofstra University in Hemps...      1
1  CLINTON:                                How are you, Donald?      2
2     HOLT:    Good luck to you. Well, I don't expect us to ...      3

Number of chunks by speaker:
Speaker
TRUMP:       123
HOLT:         97
CLINTON:      87
Name: count, dtype: int64


Preview of 2020 ['TRUMP', 'BIDEN'] debate data frame:
    Speaker                                                Text  Order
0  WALLACE:   Good evening from the Health Education Campus...      1
1    BIDEN:                                How you doing, man?      2
2    TRUMP:                                 How are you doing?      3

Number of chunks by speaker:
Speaker
TRUMP:      337
BIDEN:      266
WALLACE:    245
Name: count, dtype: int64
```

We can see this structure is in reasonable form.

b.

Now I show the results of the get_words method from earlier to get all of the individual words spoken by any speaker.

```
for i in debates:
  print(f"Preview of {i.year} {i.candidates} Debate:")
  for j in i.words.keys():
    print(f"{j}\n{i.words[j][:10]}")
  print("\n")
```

```
Preview of 2000 ['GORE', 'BUSH'] Debate:
MODERATOR
['Good', 'evening', 'from', 'the', 'Clark', 'Athletic', 'Center', 'at', 'the', 'University']
GORE
['Well', 'Jim', 'first', 'of', 'all', 'I', 'would', 'like', 'to', 'thank']
BUSH
['Well', 'we', 'do', 'come', 'from', 'different', 'places', 'I', 'come', 'from']


Preview of 2004 ['KERRY', 'BUSH'] Debate:
KERRY
['Well', 'first', 'of', 'all', 'Bob', 'thank', 'you', 'for', 'moderating', 'tonight']
BUSH
['Bob', 'thank', 'you', 'very', 'much', 'I', 'want', 'to', 'thank', 'Arizona']
SCHIEFFER
['Good', 'evening', 'from', 'Arizona', 'State', 'University', 'in', 'Tempe', 'Arizona', 'I'm']


Preview of 2008 ['MCCAIN', 'OBAMA'] Debate:
MCCAIN
['Well', 'let', 'let', 'me', 'say', 'Bob', 'thank', 'you', 'And', 'thanks']
OBAMA
['Well', 'first', 'of', 'all', 'I', 'want', 'to', 'thank', 'Hofstra', 'University']
SCHIEFFER
['Good', 'evening', 'And', 'welcome', 'to', 'the', 'third', 'and', 'last', 'presidential']


Preview of 2012 ['OBAMA', 'ROMNEY'] Debate:
OBAMA
['Well', 'thank', 'you', 'very', 'much', 'Jim', 'for', 'this', 'opportunity', 'I']
LEHRER
['Good', 'evening', 'from', 'the', 'Magness', 'Arena', 'at', 'the', 'University', 'of']
ROMNEY
['Thank', 'you', 'Jim', 'It's', 'an', 'honor', 'to', 'be', 'here', 'with']


Preview of 2016 ['TRUMP', 'CLINTON'] Debate:
TRUMP
['Thank', 'you', 'Lester', 'Our', 'jobs', 'are', 'fleeing', 'the', 'country', 'They're']
CLINTON
```

```
['How', 'are', 'you', 'Donald', 'Well', 'thank', 'you', 'Lester', 'and', 'thanks']
HOLT
['Good', 'evening', 'from', 'Hofstra', 'University', 'in', 'Hempstead', 'New', 'York', 'I'm']


Preview of 2020 ['TRUMP', 'BIDEN'] Debate:
WALLACE
['Good', 'evening', 'from', 'the', 'Health', 'Education', 'Campus', 'of', 'Case', 'Western']
TRUMP
['How', 'are', 'you', 'doing', 'Thank', 'you', 'very', 'much', 'Chris', 'I']
BIDEN
['How', 'you', 'doing', 'man', 'I'm', 'well', 'Well', 'first', 'of', 'all']
```

So for each speaker in each debate we have a list of all the words they spoke.

c.

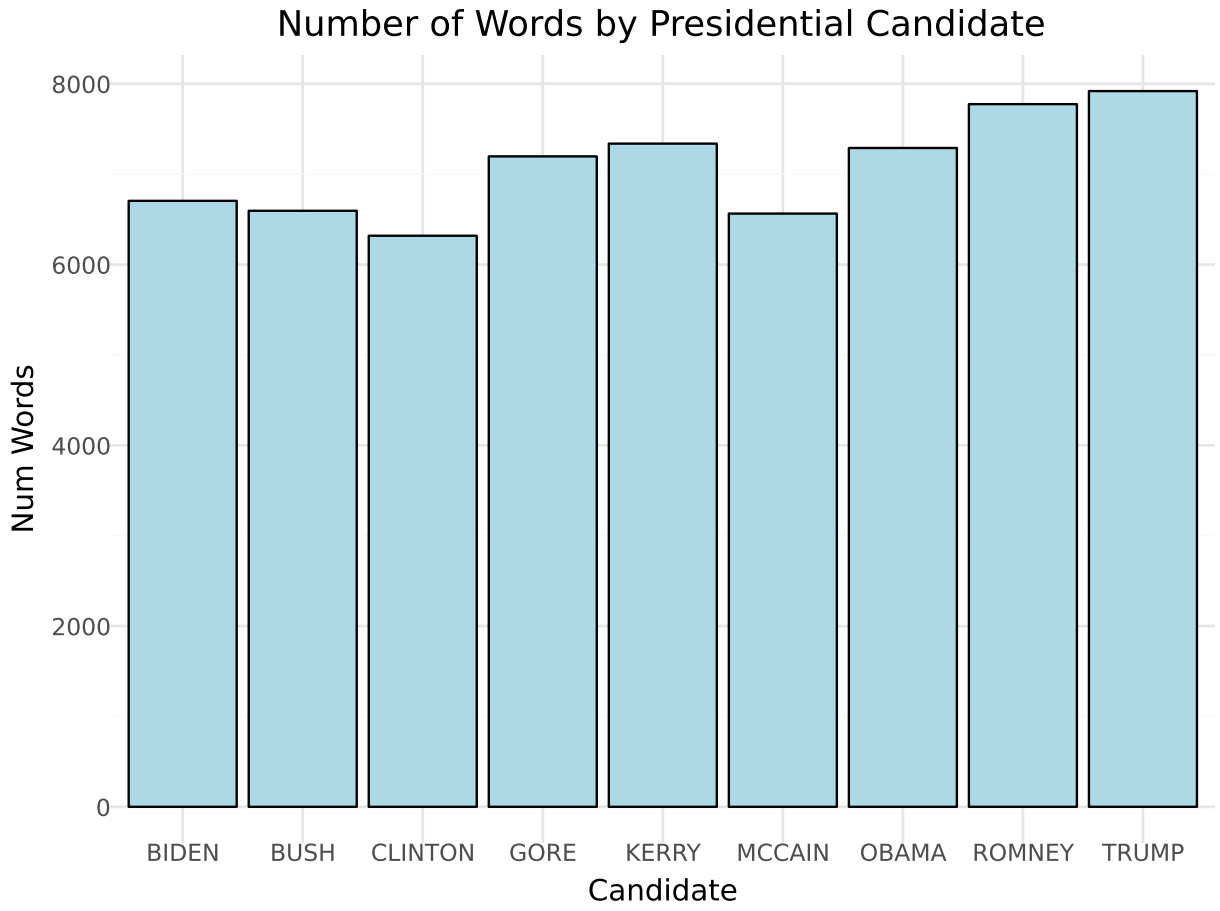
Now I show the results of the get_words method from earlier to get the desired statistics of the words spoken by candidates.

```
for i in debates:
  print(f"{i.year} {i.candidates} Debate:")
  print(i.words_stats)
  print("\n")
```

```
2000 ['GORE', 'BUSH'] Debate:
  Candidate  Num Words  Num Chars  Avg Chars  Year
0      GORE       7197      31519   4.379464  2000
1      BUSH       7441      32312   4.342427  2000


2004 ['KERRY', 'BUSH'] Debate:
  Candidate  Num Words  Num Chars  Avg Chars  Year
0     KERRY       7338      31644   4.312347  2004
1      BUSH       5749      25610   4.454688  2004


2008 ['MCCAIN', 'OBAMA'] Debate:
  Candidate  Num Words  Num Chars  Avg Chars  Year
0    MCCAIN       6564      29109   4.434644  2008
1     OBAMA       7306      32802   4.489734  2008


2012 ['OBAMA', 'ROMNEY'] Debate:
  Candidate  Num Words  Num Chars  Avg Chars  Year
0     OBAMA       7274      32608   4.482816  2012
1    ROMNEY       7775      33905   4.360772  2012


2016 ['TRUMP', 'CLINTON'] Debate:
  Candidate  Num Words  Num Chars  Avg Chars  Year
0     TRUMP       8458      36253   4.286238  2016
1   CLINTON       6319      27595   4.366988  2016


2020 ['TRUMP', 'BIDEN'] Debate:
  Candidate  Num Words  Num Chars  Avg Chars  Year
0     TRUMP       7381      30901   4.186560  2020
1     BIDEN       6705      27965   4.170768  2020
```

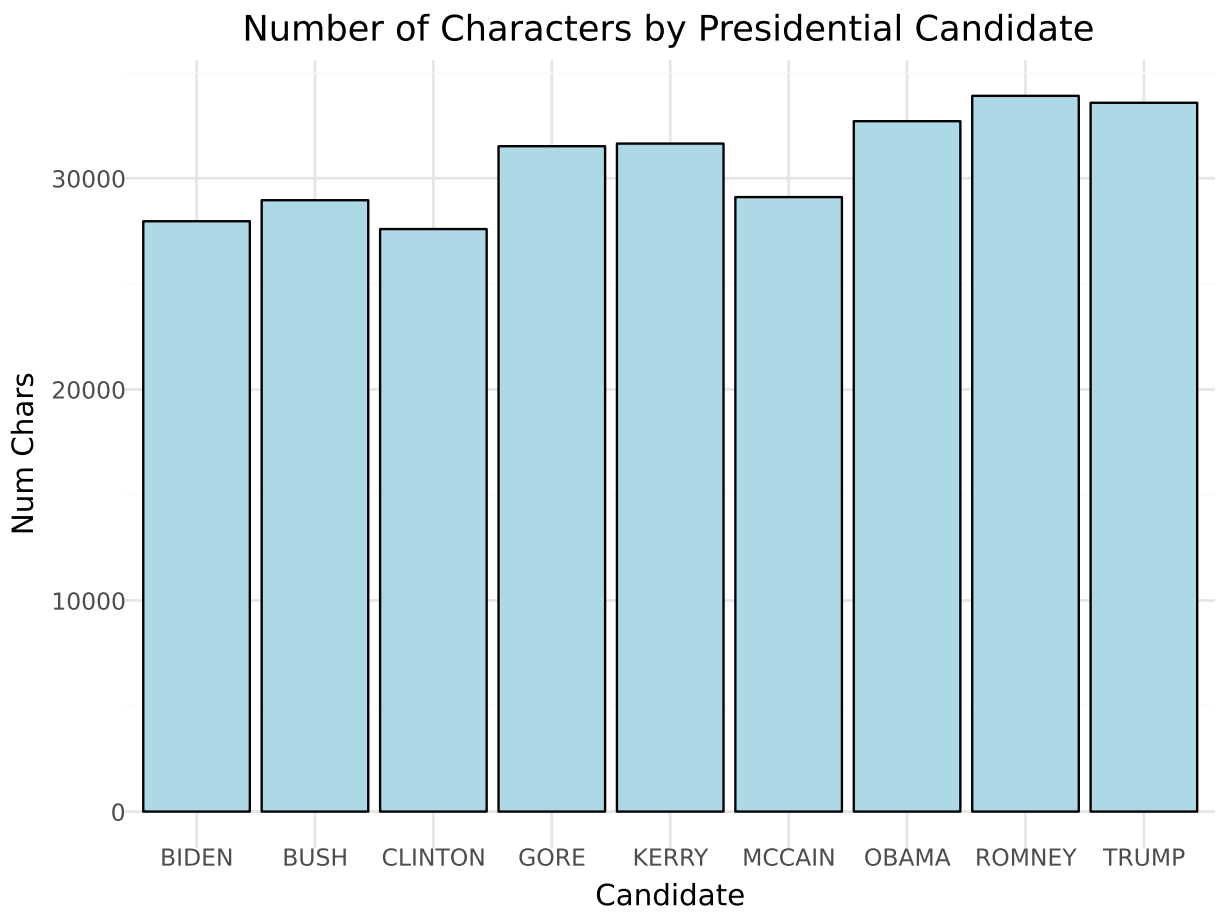
```
merged_df = debates[0].words_stats
for i in debates[1:]:
  merged_df = pd.concat([merged_df,i.words_stats],axis=0)
merged_df.index = range(len(merged_df))

# Graph results
```

```
(
ggplot(merged_df.groupby("Candidate").mean().reset_index(),
       aes(x = "Candidate")
      ) +
      geom_col(aes(y = "Num Words"),
               color = "black",
               fill = "lightblue"
              ) +
      # Relabel graph
      labs(title = "Number of Words by Presidential Candidate") +
      # Use a simplistic theme
      theme_minimal()
).show()
```

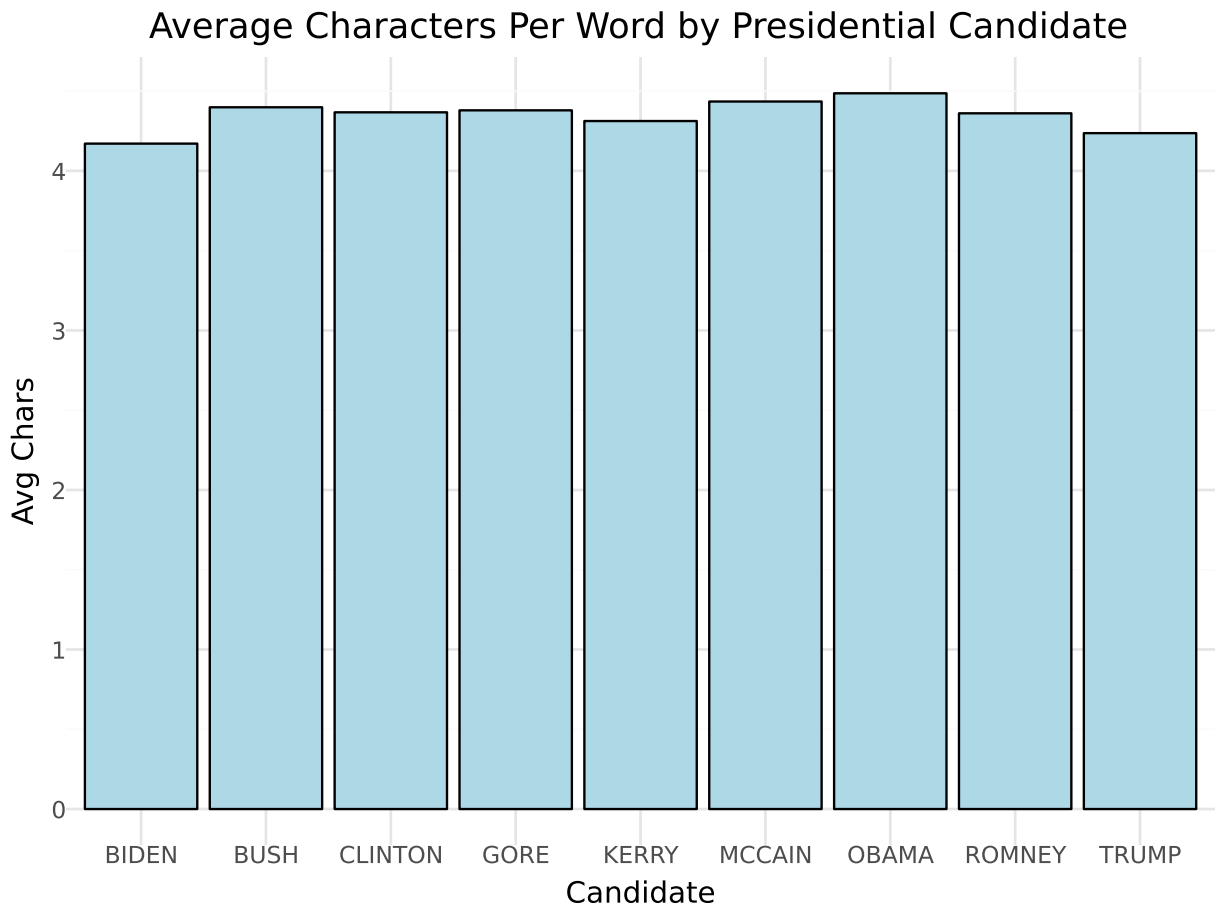



```
(
ggplot(merged_df.groupby("Candidate").mean().reset_index(),
       aes(x = "Candidate")
      ) +
      geom_col(aes(y = "Num Chars"),
               color = "black",
               fill = "lightblue"
              ) +
      # Relabel graph
      labs(title = "Number of Characters by Presidential Candidate") +
      # Use a simplistic theme
```

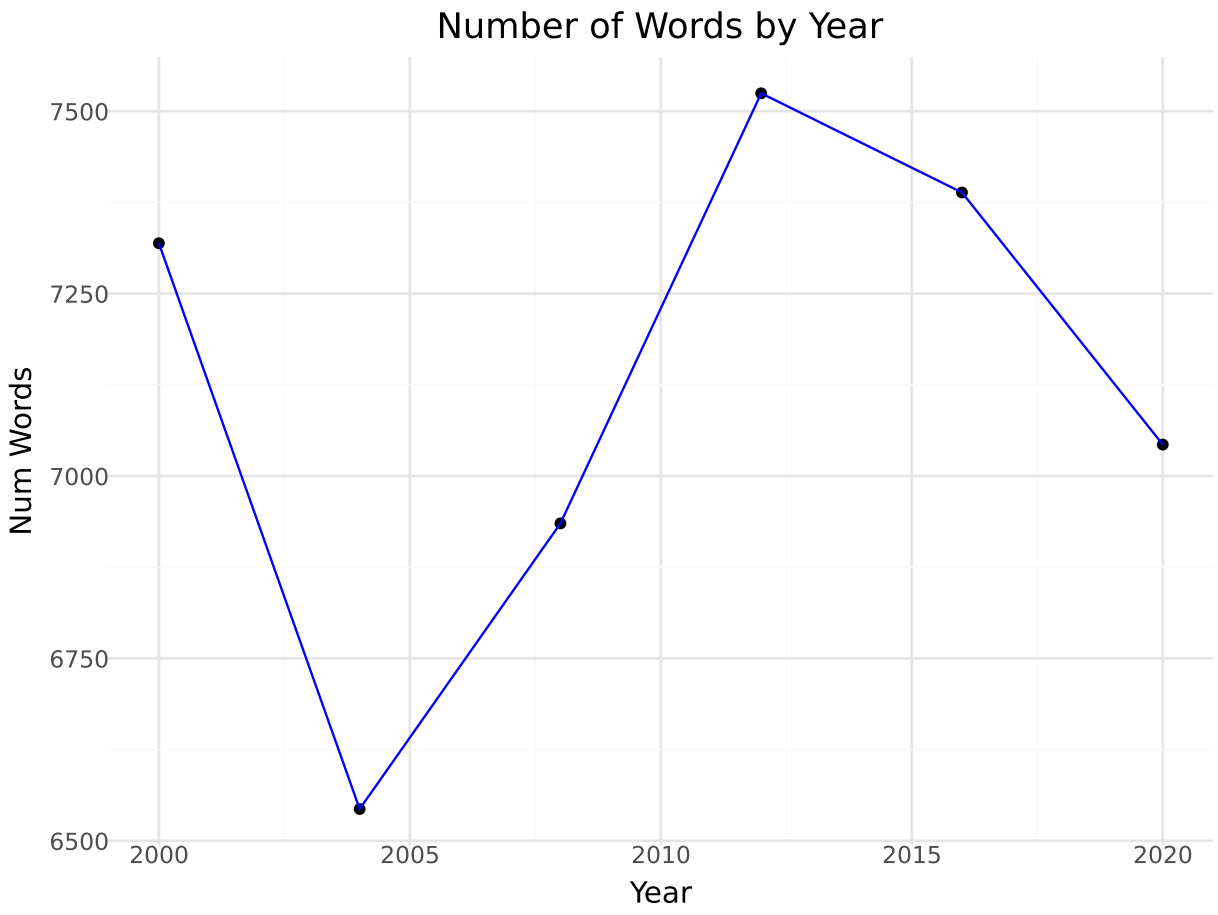
```
      theme_minimal()
).show()
```



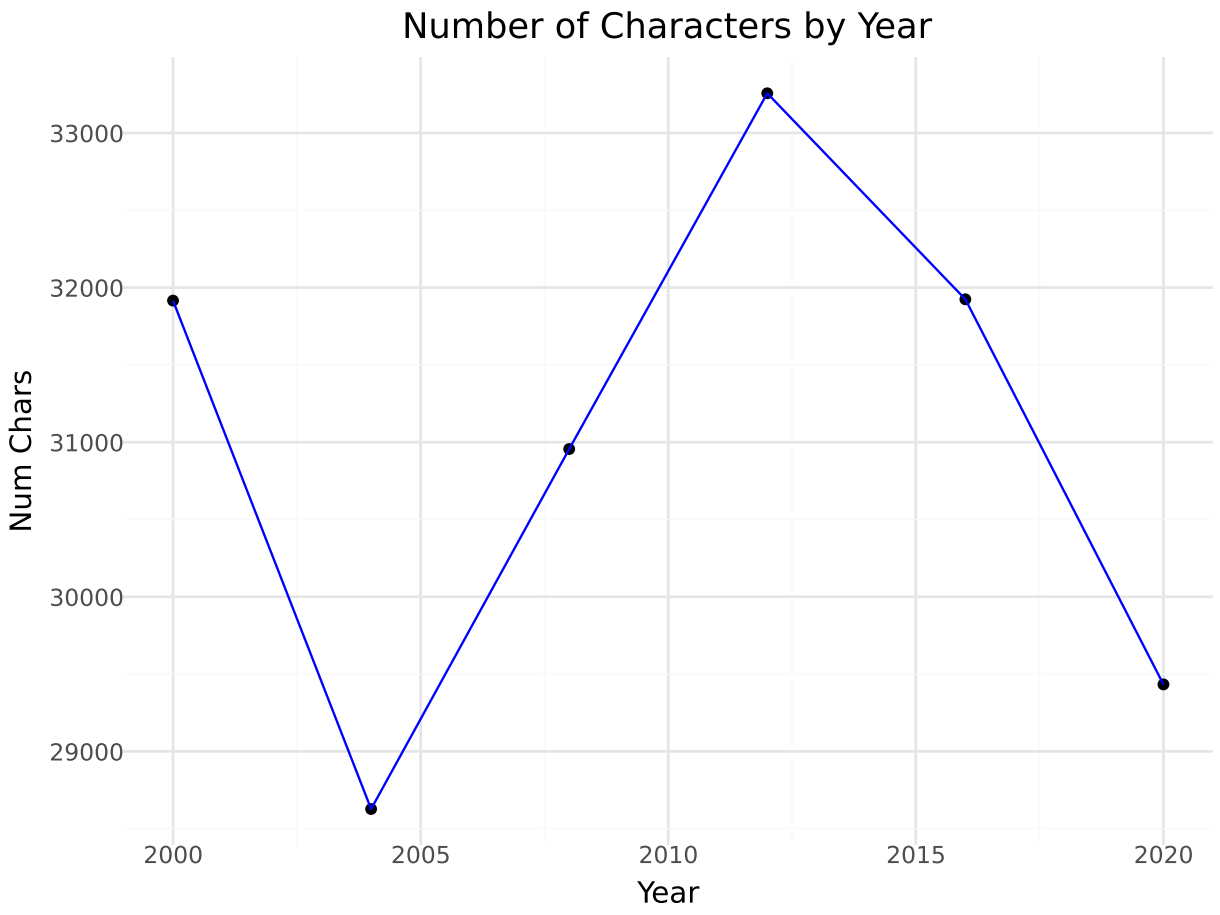

```
(
ggplot(merged_df.groupby("Candidate").mean().reset_index(),
       aes(x = "Candidate")
      ) +
      geom_col(aes(y = "Avg Chars"),
               color = "black",
               fill = "lightblue"
              ) +
      # Relabel graph
      labs(title = "Average Characters Per Word by Presidential Candidate") +
      # Use a simplistic theme
      theme_minimal()
).show()
```



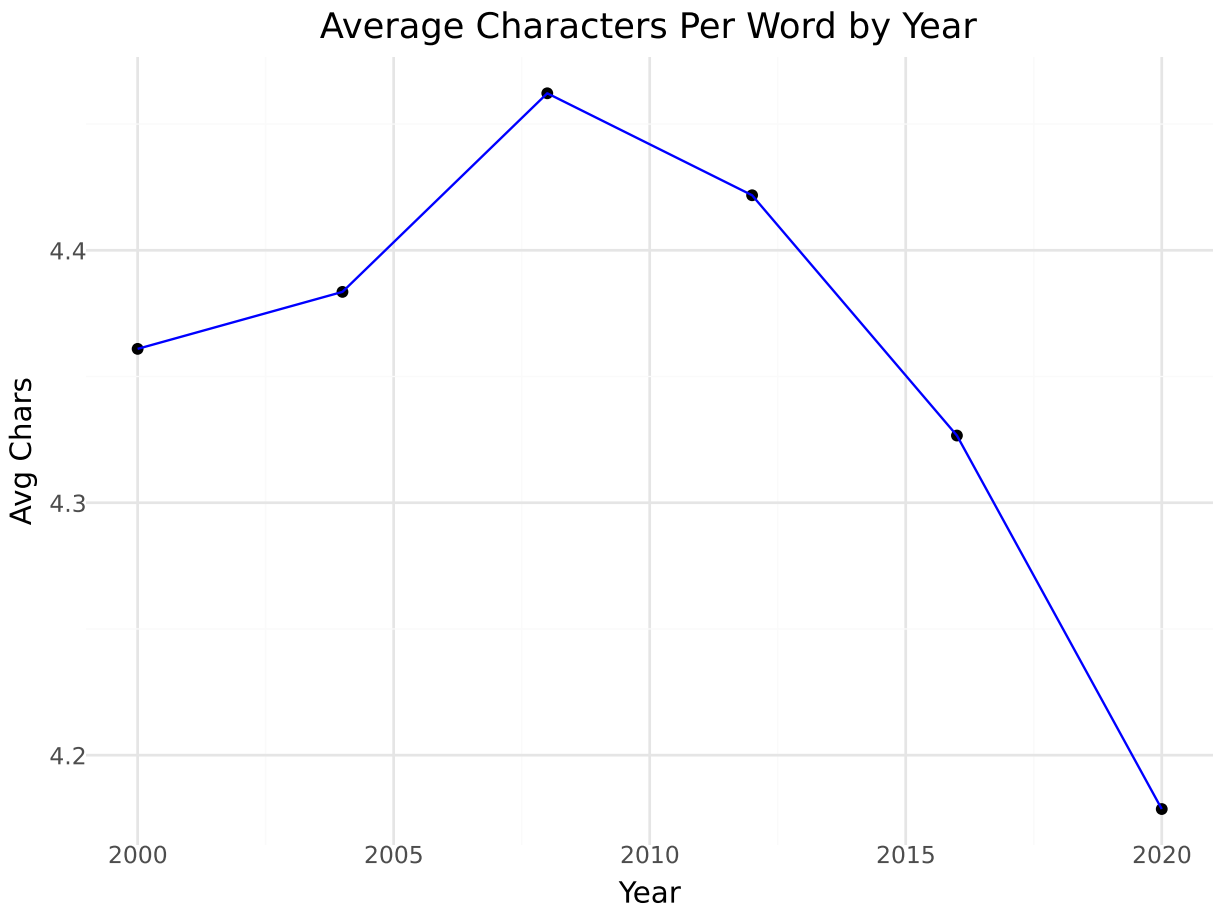

```
(
ggplot(merged_df.drop("Candidate", axis=1).groupby("Year").mean().reset_index(),
       aes(x = "Year")
      ) +
      geom_point(aes(y = "Num Words"),
                 color = "black",
                ) +
      geom_line(aes(y = "Num Words"),
                 color = "blue",
                ) +
      # Relabel graph
      labs(title = "Number of Words by Year") +
      # Use a simplistic theme
      theme_minimal()
).show()
```




```
(
ggplot(merged_df.drop("Candidate", axis=1).groupby("Year").mean().reset_index(),
       aes(x = "Year")
      ) +
      geom_point(aes(y = "Num Chars"),
                 color = "black",
                ) +
      geom_line(aes(y = "Num Chars"),
                 color = "blue",
                ) +
      # Relabel graph
      labs(title = "Number of Characters by Year") +
      # Use a simplistic theme
      theme_minimal()
).show()
```




```
(
ggplot(merged_df.drop("Candidate", axis=1).groupby("Year").mean().reset_index(),
       aes(x = "Year")
      ) +
      geom_point(aes(y = "Avg Chars"),
                 color = "black",
                ) +
      geom_line(aes(y = "Avg Chars"),
                 color = "blue",
                ) +
      # Relabel graph
      labs(title = "Average Characters Per Word by Year") +
      # Use a simplistic theme
      theme_minimal()
).show()
```


Average Characters Per Word by Year
Avg Chars
4.4
4.3
4.2
2000
2005
2010
2015
2020
Year

d.

I will just create some simple inputs to test the class.

```
# This should work
test = ChunkedDebate("1999 Random Info A: Welcome B: Hello C: Hi", [{"Dem": "A", "Rep": "B"}], ["C"]
if test is not None:
  print("Passed")
else:
  print("Failed")
```

```
Passed
```

```
try:
  ChunkedDebate("19 Random Info A: Welcome B: Hello C: Hi", [{"Dem": "A", "Rep": "B"}], ["C"])
  print("Failed")
except ValueError:
  print("Passed")
```

```
Passed
```

```
try:
  ChunkedDebate([], [], [])
  print("Failed")
except TypeError:
  print("Passed")
```

```
Passed
```

```
try:
  ChunkedDebate("", [], [])
  print("Failed")
except ValueError:
  print("Passed")
```

```
Passed
```

```
try:
  ChunkedDebate("", ["A"], [])
  print("Failed")
except TypeError:
  print("Passed")
```

```
Passed
```

```
try:
  ChunkedDebate("", [{"Dem": "A"}], [])
  print("Failed")
except ValueError:
  print("Passed")
```

```
Passed
```

```
try:
  ChunkedDebate("", [{"Dem": "A", "Rep": "B"}], "A")
  print("Failed")
except TypeError:
  print("Passed")
```

```
Passed
```

```
test.year
```

```
1999
```

```
test.candidates
```

```
['A', 'B']
```

```
test.matches
```

```
  Speaker          Text  Order
0     A:       Welcome       1
1     B:    Hello C: Hi      2
```

```
test.words
```

```
{'A': ['Welcome'], 'B': ['Hello', 'C', 'Hi']}
```

```
test.words_stats
```

```
  Candidate  Num Words  Num Chars  Avg Chars  Year
0         A          1          7   7.000000  1999
1         B          3          8   2.666667  1999
```

```
test.chunk_counts
```

```
Speaker
A:     1
B:     1
Name: count, dtype: int64
```

As we can see all of our simple tests passed, checking for possible invalid inputs and the simple case has all of the desired attributes.

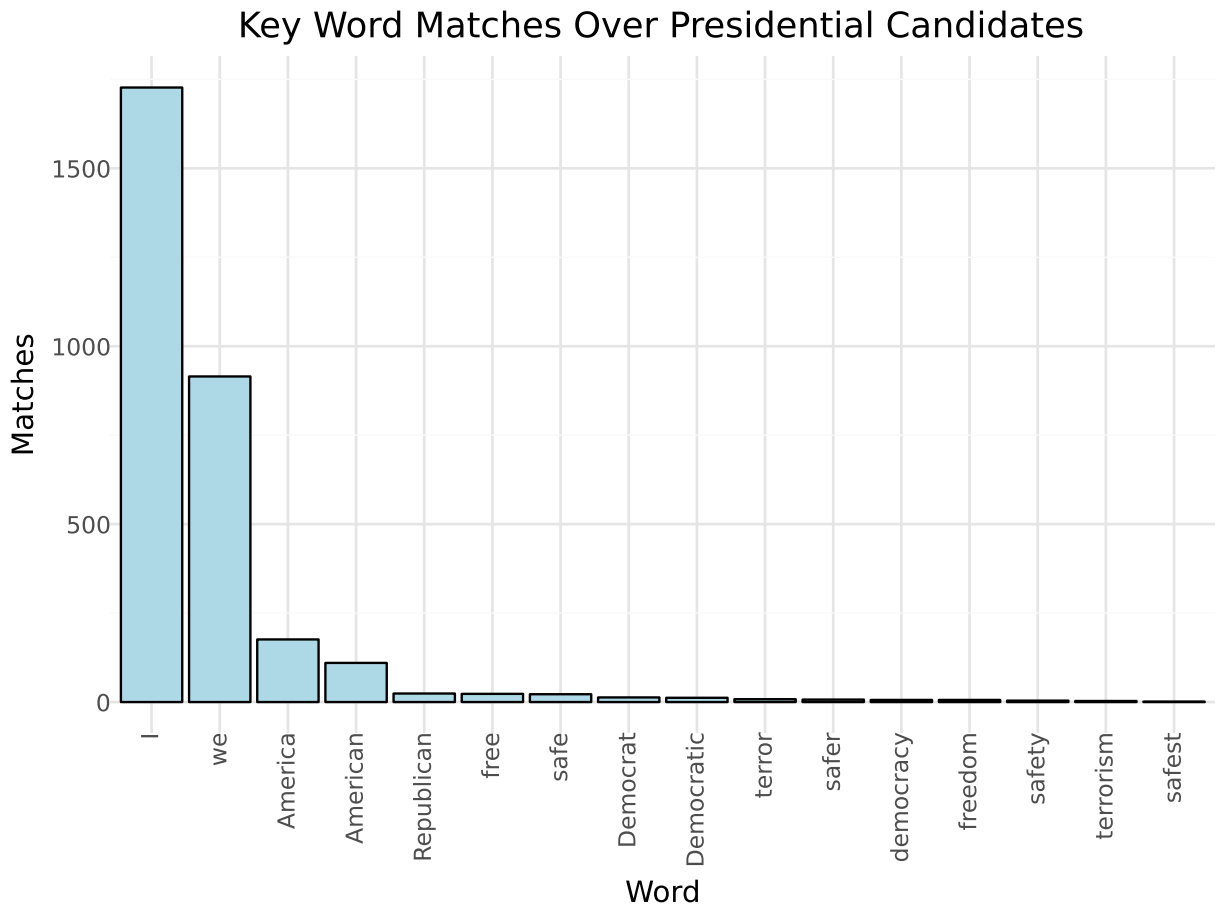
e.

```
# Regex for each word
regexes = [
  "I", "we", "American?", "democra(cy|tic)", "republic",
  "Democrat(ic)?", "Republican", "free(dom)?", "terror(ism)?",
  "safe(r|st|ty)?", "Jesus", "Christ", "Christian"
  ]
# Indicate we only want the word to take the whole string
regexes = ["^" + i + "$" for i in regexes]
df = {"Candidate": [], "Word": [], "Matches": []}
for i in debates:
  # Get candidates
  cans = i.candidates
  for j in regexes:
    # Get the matches of the regex
    words1 = [re.search(j, x).group() for x in i.words[cans[0]] if re.search(j, x) is not None]
    words2 = [re.search(j, x).group() for x in i.words[cans[1]] if re.search(j, x) is not None]
    # Add an entry for each candidate for each of the word(s) that are matched
    for k in list(set(words1)):
      df["Word"].append(k)
      df["Candidate"].append(cans[0])
      df["Matches"].append(len([word for word in words1 if word == k]))
    for k in list(set(words2)):
      df["Word"].append(k)
      df["Candidate"].append(cans[1])
      df["Matches"].append(len([word for word in words2 if word == k]))

# Make dataframe
df = pd.DataFrame(df).sort_values(by = "Matches", ascending = False)
df.index = range(len(df))
df
```

```
    Candidate        Word  Matches
0       TRUMP           I      229
1        GORE           I      195
2       KERRY           I      175
3        BUSH           I      172
4       OBAMA           I      156
..        ...         ...      ...
104   CLINTON   democracy        1
105     TRUMP    Democrat        1
106   CLINTON  Republican        1
107     KERRY     freedom        1
108      BUSH        safe        1

[109 rows x 3 columns]
```

```
(
```

```
# Graph results
ggplot(df.drop("Candidate",axis=1).groupby("Word").sum().reset_index(),
       aes(x = "reorder(Word, -Matches)")
      ) +
      geom_col(aes(y = "Matches"),
               color = "black",
               fill = "lightblue"
              ) +
      # Relabel graph
      labs(title = "Key Word Matches Over Presidential Candidates",
          x = "Word"
          ) +
      # Use a simplistic theme
      theme_minimal() +
      theme(axis_text_x = element_text(angle = 90))
).show()
```



As we can see "I" is the most used words, probably because the candidates are trying to sell themselves and are saying things like "I would do this" and so on. Then we is the next most used so they are likely trying to create a sense of unity with those watching. We can also see that they say America and American a lot indicating they are trying to show dedication to our country. The other words aren't mentioned nearly as much but we can see each major political party is mentioned and other words talking about the safety and freedom of America are mentioned.

3.

I did an OOP approach before so now I will outline a FP approach.

The approach here is essentially going to be analogous, we are going to basically use the same code for each function (with a few added functions because I want each to only return one thing) we saw before that each take in needed inputs that were previously saved as class attributes before. So this means the code of the function remains the same we are just replacing self.string for example with string as one of the input arguments.

Here are the functions we will need and a little about what they do (note I am keeping the names the same as the methods before to show how analogous this is):

- get_candidates_list
  - Purpose: Makes a list of all the candidates
  - Inputs: Dictionary containing all candidates (with keys "Dem" and "Rep")
  - Output: List of all the candidates (just their names not party)
- get_speakers
  - Purpose: Makes a list of all the speakers (moderators and candidates)
  - Inputs: List of all candidates, List of all moderators
  - Output: List of all speakers (each only appearing once)
- speaker_regex
  - Purpose: Used to construct a regex that will match any "SPEAKER:" with a possible space at the end
  - Inputs: List of all speakers
  - Output: String containing regex to match any "SPEAKER:" with a possible space at the end
- get_year
  - Purpose: Gets the year the debate was held in
  - Inputs: String (that must contain a 4 digit number in it)
  - Output: Integer of the year found
- start_location_finder
  - Purpose: Gets the start positions of all matches found with a regex
  - Inputs: String to find matches in, String with match regex
  - Output: List of all start positions for the matches
- end_location_finder
  - Purpose: Gets the end positions of all matches found with a regex
  - Inputs: String to find matches in, String with match regex
  - Output: List of all end positions for the matches
- group_finder
  - Purpose: Gets the group (i.e. what was matched) of all matches found with a regex
  - Inputs: String to find matches in, String with match regex

  - Output: List of all groups for the matches (not a unique list, items might appear multiple times)
- combine_same_speaker
  - Purpose: Makes a dictionary with chunk location information to later use to exctract chunks
  - Inputs: List of start positions from regex match, List of end positions from regex match, List of groups from regex match
  - Output: Dictionary containing start and end locations as well as group information for all matches (combining adjacent matches from same group)
- deb_info_extractor
  - Purpose: Uses chunk location information to exctract pre match info
  - Inputs: Dictionary containing start and end locations as well as group information for all matches, String that matches were found from
  - Output: String containing text found before first match
- text_exctractor
  - Purpose: Uses chunk location information to exctract chunk text, speaker, and order spoken and hold in pandas dataframe
  - Inputs: Dictionary containing start and end locations as well as group information for all matches, String that matches were found from
  - Output: Pandas dataframe with the match, text between the match and the next match, and order of the match
- IdentifyModerator:
  - Purpose: Uses chunk match information to determine if the moderator is announced at the start by "MODERATOR:"
  - Inputs: Pandas dataframe with match information
  - Output: Boolean that is true if there is only one entry for "MODERATOR:"
- adjust_deb_info:
  - Purpose: Add the moderator specification to the debate info if it was counted as a match before
  - Inputs: Boolean, String of pre match text
  - Output: String of pre match text (with the moderator specification added if the Boolean is true)
- adjust_matches:
  - Purpose: Remove the moderator specification from the matches if it was counted as a match before
  - Inputs: Boolean, Pandas dataframe of match information
  - Output: Pandas dataframe of matches (removing the moderator entry if the Boolean is true)
- speaker_counts:
  - Purpose: Count the number of chunks per speaker
  - Inputs: Pandas dataframe of match information (adjusted if needed)
  - Output: Count type object with a column for speakers and a column for number of chunks
- get_words:

  - Purpose: Gets all of the individual words spoken by each speaker
  - Inputs: Pandas dataframe of match information (adjusted if needed)
  - Output: Dictionary with columns as speakers where the entries are a list of all words spoken by that speaker

- get_word_stats:
  - Purpose: Get the number of words/characters and avg characters for each candidate
  - Inputs: Dictionary of individual words spoken by each speaker, Integer of the year of the debate
  - Output: Pandas dataframe with candidate, the number of words, number of characters, average characters per word, and year of the debate