Dependency Parsing is the process to analyze the grammatical structure in a sentence and find out related words as well as the type of the relationship between them.

Each relationship:

Has one head and a dependent that modifies the head. Is labeled according to the nature of the dependency between the head and the dependent. These labels can be found at Universal Dependency Relations.

spaCy is an open-source Python library for Natural Language Processing.

To get started, first install spaCy and load the required language model.

en_core_web_sm is the smallest English model available in spaCy with a size of 12MB. Refer spaCy English Models to view other available models.

```
import spacy
nlp=spacy.load("en_core_web_sm")

type(nlp)

# sentence="The&cat&chased&the&mouse."
# doc=nlp(sentence)
# print(doc)

sentence="The cat chased the mouse."
doc=nlp(sentence)
print(doc)
print("_____________")
count=0
for token in doc:
  print(token)
  count+=1
print(count)

sentence="The cat chased the mouse."
doc=nlp(sentence)

for token in doc:
  print(token.text, token.dep_, token.text)

print(spacy.explain("nsubj"))
```

u can assume "cat" to be noun (nominal subject)

```
print(spacy.explain("dobj"))

from spacy import displacy
displacy.render(doc, style='dep', jupyter=True, options={'distance':
120})
```

```
sentence="John reads a book in the library."
doc=nlp(sentence)

for token in doc:
  print(token.text, token.dep_, token.text)

dobj_explanation=spacy.explain("pobj")
dobj_explanation

displacy.render(doc, style='dep', jupyter=True, options={'distance':
120})
```

spaCy also provides a built-in dependency visualizer called displaCy that you can use to generate dependency graph for sentences.

displacy.render() function will generate the visualization for the sentence.

Note: If you are running the code in Jupyter Notebook or Google Colab, use jupyter = True in render() function.

```
import spacy
from spacy import displacy

# Load the language model
nlp = spacy.load("en_core_web_sm")

sentence = 'Deemed universities charge huge fees'

# nlp function returns an object with individual token information,
# linguistic features and relationships
doc = nlp(sentence)

print ("{:<15} | {:<8} | {:<15} |
{:<20}".format('Token','Relation','Head', 'Children'))
print ("-" * 70)

for token in doc:
  # Print the token, dependency nature, head and all dependents of the
token
  print ("{:<15} | {:<8} | {:<15} | {:<20}"
         .format(str(token.text), str(token.dep_),
str(token.head.text), str([child for child in token.children])))

 # Use displayCy to visualize the dependency
displacy.render(doc, style='dep', jupyter=True, options={'distance':
120})

spacy.explain("amod")
```

#Using stanza

Stanford NLP Group have also developed Stanza. It provides a Neural Network NLP Pipeline that can be customized and a Python wrapper over Stanford CoreNLP package, making it easier to use the CoreNLP features without downloading the jar files.

```
!pip install stanza
```

The next step is to import stanza and download the required language model. You can view all the available models here.

```
import stanza
stanza.download('en')
```

Initialize the neural pipeline using stanza.Pipeline() function. The first parameter is the language to use. An optional parameter processors can be passed which can be a dictionary or a comma separated string to configure the processors to use in the pipeline.

```
# en-> English/ hi-> hindi
# what\'s up?
# mwt: what is..

nlp = stanza.Pipeline('en',
                      processors = 'tokenize,mwt,pos,lemma,depparse')
```

You can find the list of all processors on Pipeline and Processors. Some of the processors might require to be preceded by some other processor in the pipeline, else they won't work. For example, pos processor requires tokenizeand mwt processors, so we need to use these two processors in the pipeline as well if we want to use pos processor.

We will now pass our sentence through the pipeline and store all the results in doc variable.

```
doc = nlp(sentence)

doc
```

If we print doc.sentences , we will see a list for each of the sentence that was passed through the pipeline. Each list contains result of all token information and linguistic features.

For example, each item in the list is an object with properties denoting lemma, universal part-of-speech, treebank-specific part-of-speech, morphological features, index of head, position in the sentence, etc.

```
print(doc.sentences[0])

import stanza

# Download the language model
stanza.download('en')

sentence = 'Deemed universities charge huge fees'
```

```
# Build a Neural Pipeline
nlp = stanza.Pipeline('en', processors =
"tokenize,mwt,pos,lemma,depparse")

# Pass the sentence through the pipeline
doc = nlp(sentence)

# Print the dependencies of the first sentence in the doc object
# Format - (Token, Index of head, Nature of dependency)
# Index starts from 1, 0 is reserved for ROOT
doc.sentences[0].print_dependencies()


print ("{:<15} | {:<10} | {:<15} ".format('Token', 'Relation',
'Head'))
print ("-" * 50)

# Convert sentence object to dictionary
sent_dict = doc.sentences[0].to_dict()

# iterate to print the token, relation and head
for word in sent_dict:
  print ("{:<15} | {:<10} | {:<15} "
         .format(str(word['text']),str(word['deprel']),
str(sent_dict[word['head']-1]['text'] if word['head'] > 0 else
'ROOT')))

import stanza

# Download the language model
stanza.download('en')

sentence = 'Deemed universities charge huge fees'

# Build a Neural Pipeline
nlp = stanza.Pipeline('en', processors =
"tokenize,mwt,pos,lemma,depparse")

# Pass the sentence through the pipeline
doc = nlp(sentence)

# Print the dependencies of the first sentence in the doc object
# Format - (Token, Index of head, Nature of dependency)
# Index starts from 1, 0 is reserved for

for word in sent_dict:
  print(word['text'],word['upos'], word['xpos'])

for word in sent_dict:
  print ("{:<15} | {:<10} | {:<15} "
```

```
        .format(str(word['text']),str(word['xpos']),
str(sent_dict[word['head']-1]['text'] if word['head'] > 0 else
'ROOT')))
```

#POS Tagging

```
# import of nltk
import nltk
# some further components for segementation, tokenization
nltk.download('punkt')
nltk.download('averaged_perceptron_tagger')
# download the universal tagset
nltk.download('universal_tagset')
# import the word_tokenize class
from nltk.tokenize import word_tokenize
# apply the word-tokenizer to a text string and find the POS tag
nltk.pos_tag(word_tokenize("In the present study, we examine the
outcomes of such a period of no exposure on the neurocognition of L2
grammar:"), tagset='universal')
```

```
[nltk_data] Downloading package punkt to /root/nltk_data...
[nltk_data]   Unzipping tokenizers/punkt.zip.
[nltk_data] Downloading package averaged_perceptron_tagger to
[nltk_data]     /root/nltk_data...
[nltk_data]   Unzipping taggers/averaged_perceptron_tagger.zip.
[nltk_data] Downloading package universal_tagset to /root/nltk_data...
[nltk_data]   Unzipping taggers/universal_tagset.zip.
```

```
[('In', 'ADP'),
 ('the', 'DET'),
 ('present', 'ADJ'),
 ('study', 'NOUN'),
 (',', '.'),
 ('we', 'PRON'),
 ('examine', 'VERB'),
 ('the', 'DET'),
 ('outcomes', 'NOUN'),
 ('of', 'ADP'),
 ('such', 'ADJ'),
 ('a', 'DET'),
 ('period', 'NOUN'),
 ('of', 'ADP'),
 ('no', 'DET'),
 ('exposure', 'NOUN'),
 ('on', 'ADP'),
 ('the', 'DET'),
 ('neurocognition', 'NOUN'),
 ('of', 'ADP'),
 ('L2', 'NOUN'),
 ('grammar', 'NOUN'),
 (':', '.')]
```

```
print(word_tokenize("In the present study, we examine the outcomes of
such a period of no exposure on the neurocognition of L2 grammar:"))
```

```
['In', 'the', 'present', 'study', ',', 'we', 'examine', 'the',
'outcomes', 'of', 'such', 'a', 'period', 'of', 'no', 'exposure', 'on',
'the', 'neurocognition', 'of', 'L2', 'grammar', ':']
```

#PSG Phrase Structure Grammer (PSG) Replacing a sequence of words without changing the meaning of the sentence. And that sequence of words will be considered as a constituent. Some basic rules are:

NP -> DET N VP -> V NP S -> NP VP Some specific rules:

NP -> (Det)(AdjP+) N NP -> NP (PP+) NP -> PP CP AdjP -> (AdvP) Adj AdvP -> (AdvP) Adv PP -> P NP VP -> V VP -> (AdvP+) V (AdvP+) VP -> (AdvP+) V (NP) (NP) (AdvP+) (PP+) (AdvP+) “+” sign means one or more occurance.

#Drawing Syntax Trees To draw a Syntax Tree of a sentence structure Bottom-up approach is followed. The steps are:

Assign POS tags for all the words in a sentence. Find the phrases Build the tree backward. Check the tree with the rules. For example, a syntax tree for the following sentence: Peter prefers the flight from Denver is given below:

Peter -> NP prefers -> V the -> Det flight -> N from -> P Denver -> NP NP -> Det N NP -> NP PP PP -> P NP VP -> V NP S -> NP VP

```
!pip install svgling
```

```
Collecting svgling
  Downloading svgling-0.4.0-py3-none-any.whl (23 kB)
Collecting svgwrite (from svgling)
  Downloading svgwrite-1.4.3-py3-none-any.whl (67 kB)
━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━ 67.1/67.1 kB 2.7 MB/s eta
0:00:00
```

```
import nltk
from nltk import Production, CFG
# grammar
cgrammar = nltk.CFG.fromstring("""
S -> NP VP
VP -> V NP
PP -> P NP
NP -> NP PP | Det N | 'Peter' | 'Denver'
V -> 'prefers'
P -> 'from'
N -> 'flight'
Det -> 'the'
""")
# print grammar
print(cgrammar, '\n')
sent = ['Peter', 'prefers', 'the', 'flight', 'from', 'Denver']
# Using Chart Parser
```

```
cparser = nltk.ChartParser(cgrammar)
for tree in cparser.parse(sent):
  print(tree)
# drawing trees
import svgling
svgling.draw_tree(tree)
```

```
Grammar with 11 productions (start state = S)
    S -> NP VP
    VP -> V NP
    PP -> P NP
    NP -> NP PP
    NP -> Det N
    NP -> 'Peter'
    NP -> 'Denver'
    V -> 'prefers'
    P -> 'from'
    N -> 'flight'
    Det -> 'the'

(S
  (NP Peter)
  (VP
    (V prefers)
    (NP (NP (Det the) (N flight)) (PP (P from) (NP Denver)))))
```

#Ambiguity

The man saw a boy with binoculars.

```
import nltk
a_grammar = nltk.CFG.fromstring("""
S -> NP VP
```

```
PP -> P NP
NP -> Det N | Det N PP | 'I'
VP -> V NP | VP PP
Det -> 'an' | 'my'
N -> 'elephant' | 'pajamas'
V -> 'shot'
P -> 'in'
""")
sent = ['I', 'shot', 'an', 'elephant', 'in', 'my', 'pajamas']
parser = nltk.ChartParser(a_grammar)
for tree in parser.parse(sent):
  print(tree)
```

```
(S
  (NP I)
  (VP
    (VP (V shot) (NP (Det an) (N elephant)))
    (PP (P in) (NP (Det my) (N pajamas)))))
(S
  (NP I)
  (VP
    (V shot)
    (NP (Det an) (N elephant) (PP (P in) (NP (Det my) (N pajamas))))))
```

#To draw trees

```
import os
import svgling
from nltk.tree import Tree
from nltk.draw.tree import TreeView
# using the string format
t1 = Tree.fromstring('(S(NP/I)(VP(VP(V/shot)(NP(Det/an)(N/elephant)))
(PP(P/in)(NP(Det/my)(N/pajamas)))))')
svgling.draw_tree(t1)
```

```
t2 = Tree.fromstring('(S(NP/I)(VP(V/shot)(NP(Det/an)(N/elephant)
(PP(P/in)(NP(Det/my)(N/pajamas))))))')
svgling.draw_tree(t2)
```

#**CFG** S -> S CONJ S | NP VP NP -> Det N | NP CONJ NP VP -> V NP Det -> "the" | "a" N -> "man" | "letter" | "girl" | "present" | "grandmother" | "cake" | "bread" V -> "wrote" | "bought" | "baked" CONJ -> "and"

#Parsing CFG

#PCFG

```
import nltk
from nltk import CFG
grammar = nltk.CFG.fromstring("""
S -> S CONJ S | NP VP
NP -> Det N | NP CONJ NP
VP -> V NP
Det -> "the" | "a"
N -> "man" | "letter" | "girl" | "present" | "grandmother" | "cake" |
"bread"
V -> "wrote" | "bought" | "baked"
CONJ -> "and"
""")
sr_parser = nltk.ShiftReduceParser(grammar, trace=2)
sent1 = 'the man wrote a letter and the girl bought a present'.split()
sent2 = 'the grandmother baked a cake and a bread'.split()
print('sent1:')
for tree in sr_parser.parse(sent1):
  print(tree)
print('sent2:')
for tree in sr_parser.parse(sent2):
  print(tree)
```

```
sent1:
Parsing 'the man wrote a letter and the girl bought a present'
    [ * the man wrote a letter and the girl bought a present]
  S [ 'the' * man wrote a letter and the girl bought a present]
  R [ Det * man wrote a letter and the girl bought a present]
  S [ Det 'man' * wrote a letter and the girl bought a present]
  R [ Det N * wrote a letter and the girl bought a present]
  R [ NP * wrote a letter and the girl bought a present]
  S [ NP 'wrote' * a letter and the girl bought a present]
  R [ NP V * a letter and the girl bought a present]
  S [ NP V 'a' * letter and the girl bought a present]
  R [ NP V Det * letter and the girl bought a present]
  S [ NP V Det 'letter' * and the girl bought a present]
  R [ NP V Det N * and the girl bought a present]
  R [ NP V NP * and the girl bought a present]
  R [ NP VP * and the girl bought a present]
  R [ S * and the girl bought a present]
  S [ S 'and' * the girl bought a present]
  R [ S CONJ * the girl bought a present]
  S [ S CONJ 'the' * girl bought a present]
  R [ S CONJ Det * girl bought a present]
  S [ S CONJ Det 'girl' * bought a present]
  R [ S CONJ Det N * bought a present]
  R [ S CONJ NP * bought a present]
  S [ S CONJ NP 'bought' * a present]
  R [ S CONJ NP V * a present]
  S [ S CONJ NP V 'a' * present]
  R [ S CONJ NP V Det * present]
  S [ S CONJ NP V Det 'present' * ]
  R [ S CONJ NP V Det N * ]
  R [ S CONJ NP V NP * ]
  R [ S CONJ NP VP * ]
  R [ S CONJ S * ]
  R [ S * ]
(S
  (S (NP (Det the) (N man)) (VP (V wrote) (NP (Det a) (N letter))))
  (CONJ and)
  (S
    (NP (Det the) (N girl))
    (VP (V bought) (NP (Det a) (N present)))))
sent2:
Parsing 'the grandmother baked a cake and a bread'
    [ * the grandmother baked a cake and a bread]
  S [ 'the' * grandmother baked a cake and a bread]
  R [ Det * grandmother baked a cake and a bread]
  S [ Det 'grandmother' * baked a cake and a bread]
  R [ Det N * baked a cake and a bread]
  R [ NP * baked a cake and a bread]
  S [ NP 'baked' * a cake and a bread]
  R [ NP V * a cake and a bread]
```

```
  S [ NP V 'a' * cake and a bread]
  R [ NP V Det * cake and a bread]
  S [ NP V Det 'cake' * and a bread]
  R [ NP V Det N * and a bread]
  R [ NP V NP * and a bread]
  R [ NP VP * and a bread]
  R [ S * and a bread]
  S [ S 'and' * a bread]
  R [ S CONJ * a bread]
  S [ S CONJ 'a' * bread]
  R [ S CONJ Det * bread]
  S [ S CONJ Det 'bread' * ]
  R [ S CONJ Det N * ]
  R [ S CONJ NP * ]
```

```
import nltk

#define the grammar for pcfg
pcfg_grammar = nltk.PCFG.fromstring("""
    S -> NP VP [1.0]
    PP -> P NP [1.0]
    VP -> V NP [0.7] | VP PP [0.3]
    NP -> NP PP [0.4]
    P -> 'with' [1.0]
    V -> 'saw' [1.0]
    NP -> 'astronomers' [0.1] | 'ears' [0.18] | 'saw' [0.04] | 'stars'
[0.18] | 'telescopes' [0.1]
    """)
```

```
S -> NP VP [1.0]
PP -> P NP [1.0]
VP -> V NP [0.7]
VP-> VP PP [0.3]
NP -> NP PP [0.4]
P -> 'with' [1.0]
V -> 'saw' [1.0]
NP -> 'astronomers' [0.1]
NP-> 'ears' [0.18]
NP-> 'saw' [0.04]
NP-> 'stars' [0.18]
NP->'telescopes' [0.1]
""")
```

```
str = "astronomers saw stars with ears"

from nltk.parse import pchart

parser = pchart.InsideChartParser(pcfg_grammar)

#print all possible trees, showing probability of each parse
```

```
for t in parser.parse(str.split()):
     print(t)

(S
  (NP astronomers)
  (VP (V saw) (NP (NP stars) (PP (P with) (NP ears))))) (p=0.0009072)
(S
  (NP astronomers)
  (VP (VP (V saw) (NP stars)) (PP (P with) (NP ears)))) (p=0.0006804)
```