40.係り受け解析結果の読み込み（形態素）

解答：

```
class Morph:
  def __init__(self,node):
    node=node.split("\t")
    feature=node[1].split(",")
    self.surface=node[0]
    self.base=feature[6]
    self.pos=feature[0]
    self.pos1=feature[1]

with open("./ai.ja.txt.parsed","r") as f:

  sentences=[]
  morphs=[]

  for line in f.readlines():

    if(line=="\n") or (line[0:2]=="* "):
      continue
    elif line!="EOS\n":
      morphs.append(Morph(line))
    elif len(morphs)!=0:
      sentences.append(morphs)
      morphs=[]
print(len(sentences))
for s in sentences[:7]:
  for m in s:
    print(vars(m))
```

実行結果：

```
{'surface': '人工', 'base': '人工', 'pos': '名詞', 'pos1': '一般'}
{'surface': '知能', 'base': '知能', 'pos': '名詞', 'pos1': '一般'}

{'surface': '人工', 'base': '人工', 'pos': '名詞', 'pos1': '一般'}
{'surface': '知能', 'base': '知能', 'pos': '名詞', 'pos1': '一般'}
{'surface': '（', 'base': '（', 'pos': '記号', 'pos1': '括弧開'}
{'surface': 'じん', 'base': 'じん', 'pos': '名詞', 'pos1': '一般'}
{'surface': 'こうち', 'base': 'こうち', 'pos': '名詞', 'pos1': '一般'}
{'surface': 'のう', 'base': 'のう', 'pos': '助詞', 'pos1': '終助詞'}
{'surface': '、', 'base': '、', 'pos': '記号', 'pos1': '読点'}
```

（結果の一部）

まとめ：

この問題の解き方は前の編と同じだ

41.係り受け解析結果の読み込み（文節・係り受け）

解答：

```
class Chunk:
  def __init__(self,morphs,dst):#这三个变量分别代表了什么？
    self.morphs=morphs
    self.dst=dst
    self.srcs=[]

with open("ai.ja.txt.parsed","r") as f:
  sentences=[]
  morphs=[]
  chunks=[]#组块
  dst=None

  for line in f.readlines():

    if line=="\n":
     continue

    elif line[0:2]=="* ":
     if len(morphs)!=0:
       chunks.append(Chunk(morphs,dst))
       morphs=[]

     dst=int(line.split(" ")[2][:-1])#这一行中，将"* 0 17D 1/1 0.388993",17 写入到 dst 里边，"['人工', '知能'] 17 []"也就是这里边的 17


    elif line!="EOS\n":
      morphs.append(Morph(line))#这一行比较好理解，"じん 名詞,一般,*,*,*,*,じん,ジン,ジン"，在这种情况下，直接写到 Morphs 类就可以了，并且增加到 morphs 这个列表里边

    elif len(morphs)!=0:#从 parsed 文件内容来看，这对应的是”EOS”行，意味着一整句话结束
      chunks.append(Chunk(morphs,dst))

      for i,chunk in enumerate(chunks):#通过枚举，遍历 chunk

        if chunk.dst!=-1:
         chunks[chunk.dst].srcs.append(i)#将指向的这个节点的节点保存在 srcs 变量当中

      sentences.append(chunks)

      morphs=[]
      chunks=[]
flag=0
```

```
for chunk in sentences[1]:
  print(
      flag,
      [morph.surface for morph in chunk.morphs],
      chunk.dst,
      chunk.srcs
  )
  flag+=1
```

実行結果：

```
0 ['人工', '知能'] 17 []
1 ['（', 'じん', 'こうち', 'のう', '、', '、'] 17 []
2 ['AI'] 3 []
3 ['〈', 'エーアイ', '〉', '）', 'と', 'は', '、'] 17 [2]
4 ['「', '『', '計算'] 5 []
5 ['（', '）', '』', 'という'] 9 [4]
6 ['概念', 'と'] 9 []
7 ['『', 'コンピュータ'] 8 []
8 ['（', '）', '』', 'という'] 9 [7]
9 ['道具', 'を'] 10 [5, 6, 8]
10 ['用い', 'て'] 12 [9]
11 ['『', '知能', '』', 'を'] 12 []
12 ['研究', 'する'] 13 [10, 11]
13 ['計算', '機', '科学'] 14 [12]
```

（結果の一部）

まとめ：

Parsed ファイルには 4 つのフォーマットの行があります。1 つ目は「 n」で、このフォーマットの行に遭遇したときには何もする必要はありません。2 つ目は「*0 17 D 1/1 0.388993」で、このフォーマットの行は chunk を示し、17 は chunk を指しています。この chunk のために chunk 変数を作成し、まず 17 を chunk.dst に記録し、この chunk を chunks リストに記録する必要があります。
3 つ目は「人工名詞、一般、*、*、*、*、人工、ジンコウ、ジンコ」であり、形態素分析がすでに行われており、この行を morphs 型変数に変換し、chunk に保存する必要がある。morphs の中にあります。4 つ目は「EOS」行で、この行は文の開始と終了をマークしており、len（morphs）を利用して文の終了をマークする EOS かどうかを判断する必要があります。そうであれば、chunks リストの各 chunk タイプノードのためにこのノードを指す部分ノードを探し、遍歴が完了したら、chunks リストを sentences リストに書き込む必要があります

42.係り元と係り先の文節の表示

解答：

```
def unsign(sentence):#这个函数的作用是取出块中的词的 surface 并去除记号
  unsigns=[]#设置一个空的列表，用来存放每个 chunk 里边词素

  for chunk in sentence:
    chunk_text=""#为每一个块单独创建一个字符串

    for morph in chunk.morphs:#对 chunk 中每一个词素进行判断

      if morph.pos!="記号":#如果不是记号，就将其保存下来
        chunk_text+=morph.surface#要保存 surface，不保存原型

    unsigns.append(chunk_text) #判定が終了したら、unsigns に文字列を保存する
  return unsigns
```

```
sentence_num=1#选择一个句子
unsigns=unsign(sentences[sentence_num])#将句子作为参量带入函数，能得到一个保存着 chunk 中非记号词素的列表

for i,chunk in enumerate(sentences[sentence_num]):# sentences[sentence_num]这个变量后边是一个 chunk 列表，保存着这一句话中所有 chunk 的信息
  if chunk.dst!=-1:
    print("\t".join([unsigns[i],unsigns[chunk.dst]]))#i 是当前 chunk 的在 unsigns 里边的位置，而 chunk.dst 是被指向节点的位置
```

実行結果：

```
人工知能        語
じんこうちのう  語
AI      エーアイとは
エーアイとは    語
計算    という
という  道具を
概念と  道具を
コンピュータ    という
という  道具を
道具を  用いて
用いて  研究する
知能を  研究する
```

（結果の一部）

まとめ：

Unsign の関数処理の流れ：

- 各 chunk の中の形態素を保存するための空のリストを設定します
- chunk ごとに 1 つの文字列を作成する
- chunk の形態素ごとの判断
- 記号でなければ保存
- surface を保存するには原型を保存しない
- 判定が終了したら、unsigns に文字列を保存する

43.名詞を含む文節が動詞を含む文節に係るものを抽出

解答：

```
def position(sentence):
  poses=[]

  for chunk in sentence:
    chunk_pos=[]
    for morph in chunk.morphs:
      if morph.pos!="記号":
       chunk_pos.append(morph.pos)
    poses.append(chunk_pos)

  return poses

sentence_num=1
unsigns=unsign(sentences[sentence_num])
```

```
poses=position(sentences[sentence_num])
for i,chunk in enumerate(sentences[sentence_num]):
  if chunk.dst!=-1:
    if ("名詞" in poses[i]) and ("動詞" in poses[chunk.dst]):
     print("\t".join([unsigns[i],unsigns[chunk.dst]]))
```

実行結果：

```
道具を　用いて
知能を　研究する
一分野を　　指す
知的行動を　　代わって
人間に　代わって
コンピューターに　　行わせる
研究分野とも　　される
```

（結果の一部）

まとめ：

考え方は前の問題と同じで、この問題は chunk の pos を取り出し、unsigns に順番に対応する。poses[i]に

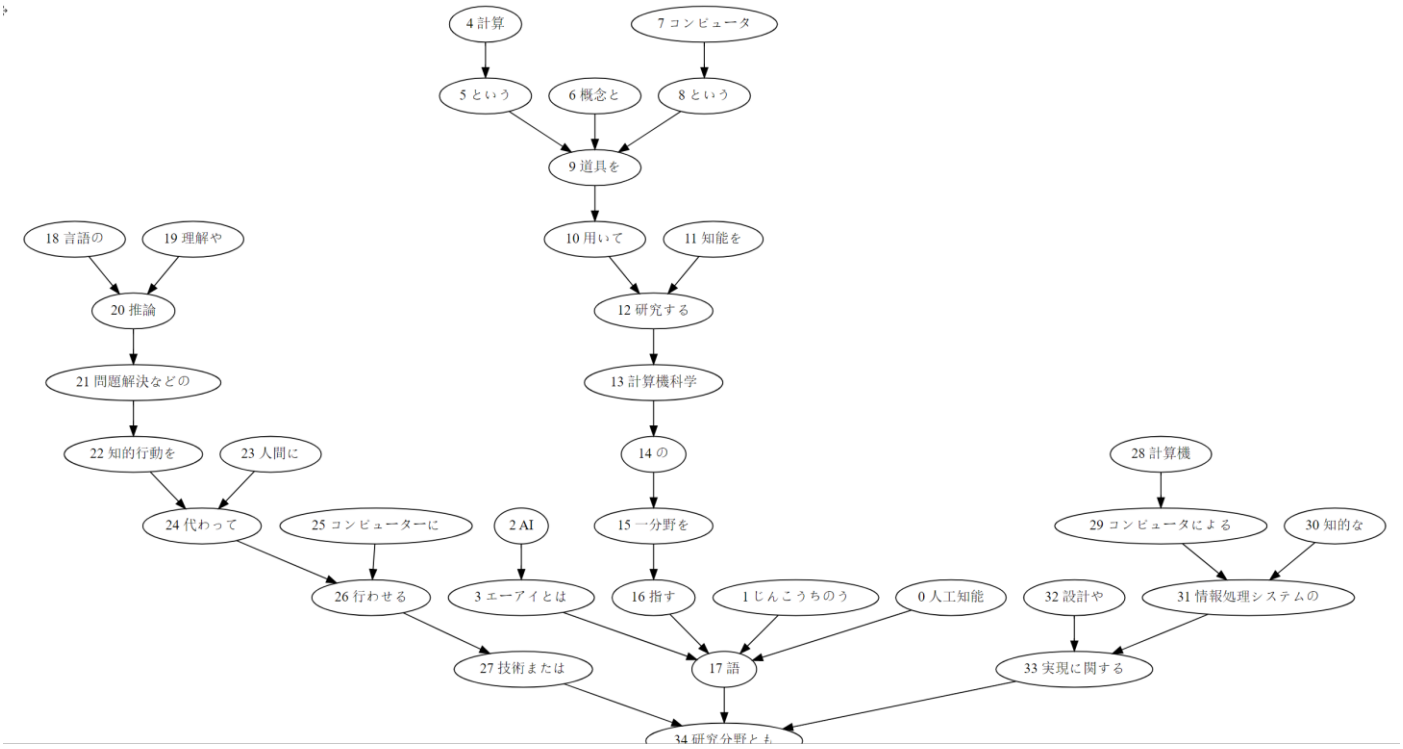
名詞があり、poses[chunk.dst]に動詞がある場合は、対応する位置の unsign を取り出します

44.係り受け木の可視化

解答：

```
from graphviz import Digraph

sentence_num=1
unsigns=unsign(sentences[sentence_num])

g=Digraph(format="png")

for i,chunk in enumerate(sentences[sentence_num]):
  if chunk.dst!=-1:
    depending=str(i)+" "+unsigns[i]
    depended=str(chunk.dst)+" "+unsigns[chunk.dst]

    g.edge(depending,depended)

g
```

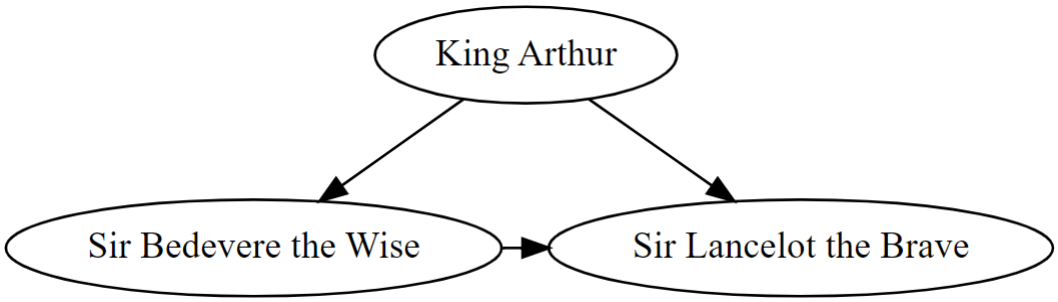
実行結果：



まとめ：

node()と edge()または edges()を使用して、図にノードとエッジを追加します。node()メソッドの最初のパラメータは、ノードの名前、オプションの label を命名するための name です。edge()メソッドのパラメータは、先頭と末尾のノード名、edges()のパラメータノードのペアです。
例えば：

```
>>> dot.node('A', 'King Arthur')
>>> dot.node('B', 'Sir Bedevere the Wise')
>>> dot.node('L', 'Sir Lancelot the Brave')

>>> dot.edges(['AB', 'AL'])
>>> dot.edge('B', 'L', constraint='false')
```



45.動詞の格パターンの抽出

解答：

```
from codecs import strict_errors
with open("./case.txt","w")as f:
  for sentence in sentences:
    for chunk in sentence:
      for morph in chunk.morphs:#检测 chunk 的词素中是否含有动词
        if morph.pos=="動詞":
```

```
            cases=[]
            for src in chunk.srcs:#寻找这个节点指向的节点是否有助词，如果有就加入到 case 当中
              case = [
                    src_morph.surface
                    for src_morph
                    in sentence[src].morphs
                    if src_morph.pos=="助詞"
                    ]
              cases.extend(case)#将 case 加入到 cases 当中
            if len(cases)>0:#如果有搜索结果将这一结果写入 case.txt
             cases=" ".join(sorted(cases))
             line=morph.base + "\t" + cases + "\n"

             f.write(line)
            break
```

実行結果：

```
用いる    を
する      て を
指す      を
代わる    に を
行う      て に
する      と も
述べる    で に の は
する      で を
する      を
する      を
する      て を
ある      が は
する      で に により
用いる    を
する      と を
使う      で でも は
呼ぶ      も
ある      て も
する      を
出す      が に
する      に を
呼ぶ      と
持つ      が に
なる      が と
```

（結果の一部）

まとめ：

処理の流れ：

- chunk の形態素に動詞が含まれているかどうかを検出する
- このノードが指すノードに助詞があるかどうかを探し、あれば case に加える
- case を cases に追加
- 検索結果があれば、その結果を case.txt に書き込みます

46.動詞の格フレーム情報の抽出

解答：

```
from codecs import strict_errors
with open("./case2.txt","w")as f:
```

```
  for sentence in sentences:

    unsigns=unsign(sentence)

    for chunk in sentence:
      for morph in chunk.morphs:
        if morph.pos=="動詞":

          cases=[]
          frames=[]

          for src in chunk.srcs:
            case = [
                 src_morph.surface
                 for src_morph
                 in sentence[src].morphs
                 if src_morph.pos=="助詞"
                 ]

            cases.extend(case)
            if len(case)>0:
             frames.append(unsigns[src])

          if len(cases)>0:
           cases=" ".join(sorted(cases))
           frames=" ".join(sorted(frames))
           line="\t".join([morph.base,cases,frames])+ "\n"
           f.write(line)
          break
```

実行結果：

```
用いる　を　道具を
する　て　を　用いて　知能を
指す　を　一分野を
代わる　に　を　人間に　知的行動を
行う　て　に　コンピューターに　代わって
する　と　も　研究分野とも
述べる　で　に　の　は　佐藤理史は　次のように　解説で
する　で　を　コンピュータ上で　知的能力を
する　を　推論判断を
する　を　画像データを
[1]する　て　を　パターンを　解析して
[1]ある　が　は　応用例は　画像認識等が
[1]する　で　に　により　1956年に　ジョンマッカーシーにより　ダートマス会議で
[1]用いる　を　記号処理を
[1]する　と　を　主体と　記述を
[1]使う　で　でも　は　意味あいでも　現在では
```

（結果の一部）

まとめ：

この問題の考え方は前の問題と同じですが、最終結果に動詞指向の chunk を追加する必要があり、42 問題で設計された unsign 関数を使用すると予想される結果を実現できることに注意してください。

47.機能動詞構文のマイニング

解答：

```
def sahen_wo(chunk):
  if len(chunk.morphs)==2:
    sahen=chunk.morphs[0].pos1=="サ変接続"
    wo=chunk.morphs[1].surface=="を"
    return sahen and wo
  return False
from codecs import strict_errors
with open("./case3.txt","w")as f:
  for sentence in sentences:
   unsigns=unsign(sentence)
   for chunk in sentence:
    for morph in chunk.morphs:
     if morph.pos=="動詞":
      for src in reversed(chunk.srcs):
       if sahen_wo(sentence[src]):
         sahen_wo_verb = unsigns[src]+morph.base
         cases=[]
         frames=[]

         for sahen_src in chunk.srcs:
          if sahen_src==src:
            continue
          case = [
               src_morph.surface
               for src_morph
               in sentence[sahen_src].morphs
               if src_morph.pos=="助詞"
              ]
          cases.extend(case)

          if len(case)>0:
           frames.append(unsigns[sahen_src])
         if len(cases)>0:
           cases=" ".join(sorted(cases))
           frames=" ".join(sorted(frames))
           line="\t".join([sahen_wo_verb,cases,frames])  + "\n"
           f.write(line)
         break
```

実行結果：

```
記述をする　と　主体と
注目を集める　が　サポートベクターマシンが
学習を行う　に　を　元に　経験を
進化を見せる　て　て　において　は　加えて　敵対的生成ネットワークは　活躍している　生成技術において
進化をいる　て　て　において　は　加えて　敵対的生成ネットワークは　活躍している　生成技術において
開発を行う　は　エイダ・ラブレスは
意味をする　に　データに
研究を進める　て　費やして
命令をする　で　機構で
運転をする　に　元に
```

（結果の一部）

まとめ：

この問題では、新しい sahen _wo 関数、この関数は chunk に「サ変接続」が含まれているかどうかを判断するために使用され、もしあれば、前の問題の方法に従ってそれを記録します

48.名詞から根へのパスの抽出

解答：

```
def root(sentence):
  unsigns=unsign(sentence)
  poses=position(sentence)
  for i,chunk in enumerate(sentence):
    if "名詞" in poses[i]:
      dst=i
      path=[]
      while dst != -1:
       path.append(unsigns[dst])
       dst=sentence[dst].dst

      print("->".join(path))
root(sentences[1])
```

実行結果：

```
人工知能->語->研究分野とも->される
じんこうちのう->語->研究分野とも->される
AI->エーアイとは->語->研究分野とも->される
エーアイとは->語->研究分野とも->される
計算->という->道具を->用いて->研究する->計算機科学->の->一分野を->指す->語->研究分野とも->される
概念と->道具を->用いて->研究する->計算機科学->の->一分野を->指す->語->研究分野とも->される
コンピューター->という->道具を->用いて->研究する->計算機科学->の->一分野を->指す->語->研究分野とも->される
道具を->用いて->研究する->計算機科学->の->一分野を->指す->語->研究分野とも->される
知能を->研究する->計算機科学->の->一分野を->指す->語->研究分野とも->される
研究する->計算機科学->の->一分野を->指す->語->研究分野とも->される
計算機科学->の->一分野を->指す->語->研究分野とも->される
一分野を->指す->語->研究分野とも->される
語->研究分野とも->される
言語の->推論->問題解決などの->知的行動を->代わって->行わせる->技術または->研究分野とも->される
理解や->推論->問題解決などの->知的行動を->代わって->行わせる->技術または->研究分野とも->される
推論->問題解決などの->知的行動を->代わって->行わせる->技術または->研究分野とも->される
問題解決などの->知的行動を->代わって->行わせる->技術または->研究分野とも->される
知的行動を->代わって->行わせる->技術または->研究分野とも->される
人間に->代わって->行わせる->技術または->研究分野とも->される
コンピューターに->行わせる->技術または->研究分野とも->される
技術または->研究分野とも->される
計算機->コンピュータによる->情報処理システムの->実現に関する->研究分野とも->される
コンピュータによる->情報処理システムの->実現に関する->研究分野とも->される
知的な->情報処理システムの->実現に関する->研究分野とも->される
情報処理システムの->実現に関する->研究分野とも->される
設計や->実現に関する->研究分野とも->される
実現に関する->研究分野とも->される
研究分野とも->される
```

まとめ：

この問題の解き方は 44 問題と似ている。まず文節を取り出し、その後各 chunk を検査し、文節で各 chunk から最終ノードへの経路を記録する。

49.名詞間の係り受けパスの抽出

解答：

```
import re

def covered_unsign(chunk,character):
  noun=[(morph.surface,morph.pos) if morph.pos !="記号
" else ("","") for morph in chunk.morphs]
  noun=[morph[0] if morph[1] !="名詞" else character for morph in noun]
  noun="".join(noun)

  charas=character + "+"
  noun=re.sub(charas, character, noun)

  return noun
def noun_path(sentence):
  unsigns=unsign(sentence)
  poses=position(sentence)
  pathes={}
  for i,chunk in enumerate(sentence):
   if "名詞" in poses[i]:
    dst=chunk.dst
    path=[]
    while dst != -1:
     path.append(dst)
     dst=sentence[dst].dst

    pathes[i]=path
  return pathes
def depend_path(sentence):
  unsigns=unsign(sentence)
  pathes=noun_path(sentence)
  for i,chunk in enumerate(sentence):
    if i in pathes.keys():
     i_noun=covered_unsign(chunk,"X")

     for j in range(i+1,len(sentence)):
       if j in pathes.keys():
         if j in pathes[i]:
           out_path=[i_noun]
           for k in pathes[i]:
            if j==k:
              j_noun=covered_unsign(sentence[j],"Y")
```

```
                out_path.append(j_noun)

                print("->".join(out_path))

            out_path.append(unsigns[k])
        else:

          for dst in pathes[j]:

            if dst in pathes[i]:
              depending_out=[i_noun]
              j_noun=covered_unsign(sentence[j],"Y")
              depended_out=[j_noun]

              for k in pathes[i]:
                if k == dst:
                  depending_out="->".join(depending_out)
                  break
                depending_out.append(unsigns[k])
              for k in pathes[j]:
                if k == dst:
                  depended_out="->".join(depended_out)
                  break
                depended_out.append(unsigns[k])
              out = " | ".join([depending_out,depended_out,unsigns[dst]])
              print(out)
              break
```

実行結果：

```
X | Yのう | 語
X | Y->エーアイとは | 語
X | Yとは | 語
X | Y->という->道具を->用いて->研究する->計算機科学->の->一分野を->指す | 語
X | Yと->道具を->用いて->研究する->計算機科学->の->一分野を->指す | 語
X | Y->という->道具を->用いて->研究する->計算機科学->の->一分野を->指す | 語
X | Yを->用いて->研究する->計算機科学->の->一分野を->指す | 語
X | Yを->研究する->計算機科学->の->一分野を->指す | 語
X | Yする->計算機科学->の->一分野を->指す | 語
X | Y->の->一分野を->指す | 語
X | Yを->指す | 語
X->Y
```

（結果の一部）

まとめ：

- ライブラリを import する
- 文節を反復させ、名詞を含む文節のインデックス番号を nouns に格納する。
- 名詞句ペアの文節番号 i、j を反復させる。

- while 文では、文節 i の構文木根に至る経路上に文節 j が存在する場合とそれ以外で条件分岐を行い、それぞれ名詞を含む文節番号とその文節の係り先文節番号を取得し、それぞれの path リストに格納する。
- 文節 i から構文木の根に至る経路上に文節 j が存在する場合を説明する。（if len(path_J == 0:）
- 文節 i と j に含まれる名詞句を X と Y に置換する。
- 残りは、文節 i と文節 j の構文木経路の間のパスを取得し、連結させて出力する。
- 文節 i から構文木の根に至る経路上に文節 j が存在しない場合を説明する。（2 つ目の else:）
- 前半の条件分岐（if len(path_J == 0:）と同様に、文節 i と j に含まれる名詞句を X と Y に置換する。
- 文節 i と文節 j の構文木の根に至る経路上で交わる文節 k を取得する。
- 文節 i と文節 j で、文節 k に至るまでのパスを取得する。
- 文節 i から文節 k に至るまでのパスと、文節 j から文節 k に至るまでのパスと、文節 k の内容をそれぞれ” | ”で連結させて、出力する。