Applications of Morphology and the concepts learnt in theory of Morphological Analysis can be used in those applications

1. Machine Translation
2. Spell Checker
3. POS Tagger
4. Information Retrieval
5. Word Sense Disambiguation

1. Machine Translation: Machine Translation is typically one of the applications that need to handle analysis and generation processes at the same time.
   Concepts from morphological analysis used here are:
   Linguistic databases for MT systems need to be designed so that the knowledge they store is as much process independent as possible. Thus designing declarative lexicon databases in MT applications is a must.
   Morphology is important to lexicon because by representing knowledge about the internal structure of words and the rules of word formation, we can save search time.

2. Information Retrieval: Information Retrieval (IR), that deals with the processing of collections of documents containing 'free text', such as scientific papers, or even the contents of electronic textbooks. The objective of such processing is to facilitate rapid and accurate search of the text based on keywords of interest.
   Concepts from morphological analysis used here are:
   Different forms of a word often communicate essentially the same meaning. These syntactic differences between word forms are called inflections, and they create challenges for query understanding.
   Stemming and lemmatization are two approaches to handle inflections in search queries.

3. Spell Checker: The purpose of spell checking is the detection and correction of typographic and orthographic errors in the text at the level of word occurrence considered out of its context.
   Concepts from morphological analysis used here are:
   Lexicon: stems and affixes (with corresponding pos)
   Morphotactics of the language: Model of how morphemes can be affixed to a stem. E.g., plural morpheme follows noun in English
   Orthographic rules: spelling modifications that occur when affixation occurs
   in il in context of l (in- + legal)

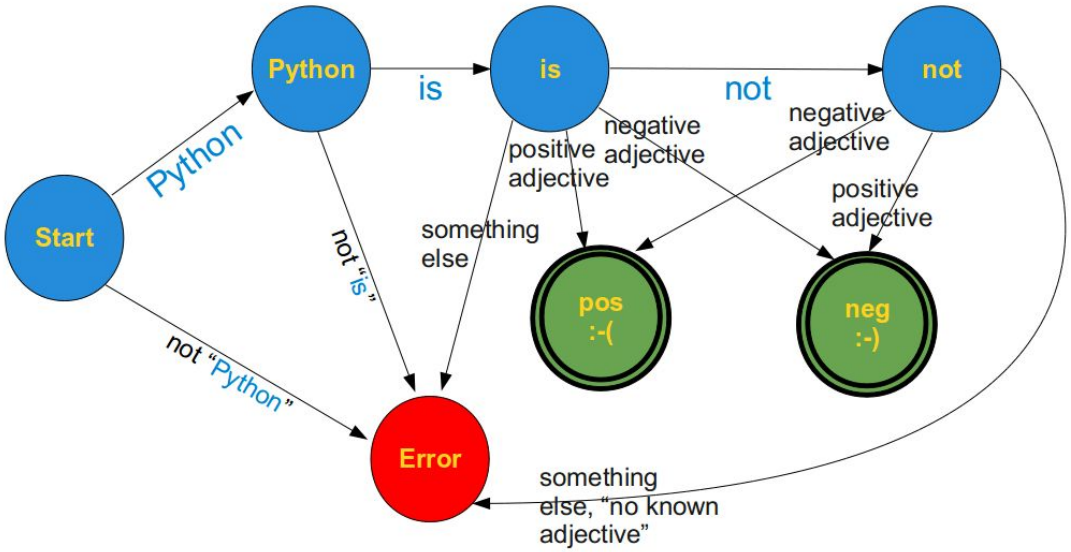
4. POS Tagger: A word can be classified into one or more lexical or parts-of-speech categories such as nouns, verbs, adjectives, and adverbs, to name a few. Parts of speech (POS) tagging is the process of labeling annotation of syntactic categories for each word in a corpus.
   Concepts from morphological analysis used here are:
   Once the data is tagged either at a morphosyntactic (uses concepts such as compounding, cliticization, morphological parsing by finding constituent morphemes and features) or POS level then the State-of-the-art POS taggers can achieve a good accuracy using statistics to determine its function.

5. Word Sense Disambiguation: Word sense disambiguation is defined as the task of finding the correct sense of a word in a context.
   Concepts from morphological analysis used here are:
   Different forms of a word often communicate essentially the same meaning. These syntactic differences between word forms are called inflections, and they create challenges for query understanding. Stemming and lemmatization are two approaches to handle inflections and morphological recognition using finite state machines will let us know whether those words really belong to a desired language.

Understand and Interpret Sample FSA code for Language recognition problem:

“Finite State Automata" is an abstract machine which consists of a set of states (including the initial state and one or more end states), a set of input events, a set of output events, and a state transition function. A transition function takes the current state and an input event as an input and returns the new set of output events and the next (new) state. Some of the states are used as "terminal states".

A Simple Example : We want to recognize the meaning of very small sentences with an extremely limited vocabulary and syntax: These sentences should start with "Python is" followed by an adjective or the word "not" followed by an adjective.
e.g.
"Python is great" → positive meaning
"Python is stupid" → negative meaning
"Python is not ugly" → positive meaning



To implement the above example, we program first a general Finite State Machine in Python. We save this class as statemachine.py
This class consists of methods add_state(), set_start(), run()

Using this general FSM, we built our program for the above automata.

Initially we have taken list of few positive and negative adjectives as follows
positive_adjectives = ["great","super", "fun", "entertaining", "easy"]
negative_adjectives = ["boring", "difficult", "ugly", "bad"]

From the main function, we have created an object ***m = StateMachine()*** and added start state to it as ***m.add_state("Start", start_transitions)*** in which start_transitions is a method which returns newState as “python_state” if the start word is Python else returns as “error state”.
we add next state as ***m.add_state("Python_state", python_state_transitions)*** in which python_state_transitions is a method which returns newState as “is_state” if the next word is “is” else returns as “error state”.
Then we add next state as ***m.add_state("is_state", is_state_transitions)*** in which is_state_transitions is a method which returns newState as “not_state” if word is "not" or returns newState as "pos_state" if word is in positive_adjectives list or returns newState as "neg_state" if word is in negative_adjectives list or returns newState as "error_state" if none of the above condition satisfies.
Then we add next state as ***m.add_state("not_state", not_state_transitions)*** in which not_state_transitions is a method which returns newState as "neg_state" if word is in positive_adjectives or returns newState as "pos_state" if word is in negative_adjectives or returns newState as "error_state" if all of the above conditions fails.
Then as the next state is going to be final state, we add
m.add_state("neg_state", None, end_state=1)
m.add_state("pos_state", None, end_state=1)
m.add_state("error_state", None, end_state=1)
Later we can try few examples as below
m.set_start("Start")
m.run("Python is great") #reached pos_state which is an end state
m.run("Python is difficult") #reached neg_state which is an end state
m.run("Perl is ugly") #reached error_state which is an end state

From these examples, we can clearly see that the designed finite state automata is recognizing the desired language.

Understand and Interpret Sample FST code for Generation problem:

As we have seen above, FSAs can recognize (accept) a string, but they don't tell us its internal structure. We need a machine that maps (transduces) the input string into an output string that encodes its structure.

Finite-state transducers (FSTs) are a generalization of FSAs, where each transition is associated with a pair of labels. Then each pair forms part of not one string but two, an input string and an output string. As a result, transducers model relations between pairs of strings. FSAs can be thought of as a special case in which every transition has the same input and output label.

Imagine that we have a collection of texts and we wish to place XML-style tags around any mention of a various types tiffins.

So, for instance, given a string like input_string = "Do you have Idli or Poori?"

And a list of tiffins as
tiffins = ("Dosa", "Idli", "Chapathi", "Vada", "Poori", "Paratha", "Uthappam")

We would produce output_string = "Do you have <tiffin>Idli</tiffin> or <tiffin>Poori</tiffin>"

However, it is easy to construct a deterministic FSA using Pynini (http://www.opengrm.org/twiki/bin/view/GRM/Pynini), one of several open-source finite-state transducer libraries developed at Google.

```
fst_target = pynini.string_map(tiffins)
```

This function constructs an FSA with a prefix tree (or trie) topology, guaranteeing that the resulting FSA will be deterministic.

With our deterministic FSA fst_target, the simplest solution is to create an FST which represents the substitution as a string-to-string rewrite relation, as follows. First, we construct transducers which insert the left and right tags; i.e., they literally map from the empty string to the tag.

```
ltag = pynini.transducer("", "<tiffin>")
rtag = pynini.transducer("", "</tiffin>")

substitution = ltag + fst_target + rtag
```

Now, this transducer represents the tag insertion relation itself. But as written, we cannot apply it to arbitrary strings, for it does not match any part of a string which is not part of a cheese name. To complete the task, we need an FST which passes through any part of a string which does not match. For this we use cdrewrite (short for "context-dependent rewrite").

```
chars = ([chr(i) for i in xrange(1, 91)] +
        ["\[", "\\", "\]"] +
        [chr(i) for i in xrange(94, 256)])
sigma_star = pynini.string_map(chars).closure()

rewrite = pynini.cdrewrite(substitution, "", "", sigma_star)
```

Then, all that remains is to apply this to a string. The simplest way to do so is to compose a string and the rewrite transducer, then convert the resulting path back into a string.

```
output = pynini.compose(input_string, rewrite).stringify()
```

Code for FSA:

Statemachine.py

```
class StateMachine:
    def __init__(self):
        self.handlers = {}
        self.startState = None
        self.endStates = []

    def add_state(self, name, handler, end_state=0):
        name = name.upper()
        self.handlers[name] = handler
        if end_state:
            self.endStates.append(name)

    def set_start(self, name):
        self.startState = name.upper()

    def run(self, cargo):
        try:
            handler = self.handlers[self.startState]
        except:
            raise InitializationError("must call .set_start() before .run()")
        if not self.endStates:
            raise  InitializationError("at least one state must be an end_state")

        while True:
            (newState, cargo) = handler(cargo)
            if newState.upper() in self.endStates:
                print("reached ", newState)
                break
            else:
                handler = self.handlers[newState.upper()]
```

Statemachine_test.py:

```
from statemachine import StateMachine

positive_adjectives = ["great","super", "fun", "entertaining", "easy"]
negative_adjectives = ["boring", "difficult", "ugly", "bad"]

def start_transitions(txt):
    splitted_txt = txt.split(None,1)
```

```
    word, txt = splitted_txt if len(splitted_txt) > 1 else (txt,"")
    if word == "Python":
        newState = "Python_state"
    else:
        newState = "error_state"
    return (newState, txt)

def python_state_transitions(txt):
    splitted_txt = txt.split(None,1)
    word, txt = splitted_txt if len(splitted_txt) > 1 else (txt,"")
    if word == "is":
        newState = "is_state"
    else:
        newState = "error_state"
    return (newState, txt)

def is_state_transitions(txt):
    splitted_txt = txt.split(None,1)
    word, txt = splitted_txt if len(splitted_txt) > 1 else (txt,"")
    if word == "not":
        newState = "not_state"
    elif word in positive_adjectives:
        newState = "pos_state"
    elif word in negative_adjectives:
        newState = "neg_state"
    else:
        newState = "error_state"
    return (newState, txt)

def not_state_transitions(txt):
    splitted_txt = txt.split(None,1)
    word, txt = splitted_txt if len(splitted_txt) > 1 else (txt,"")
    if word in positive_adjectives:
        newState = "neg_state"
    elif word in negative_adjectives:
        newState = "pos_state"
    else:
        newState = "error_state"
    return (newState, txt)

def neg_state(txt):
    print("Hallo")
```

```
    return ("neg_state", "")

if __name__== "__main__":
    m = StateMachine()
    m.add_state("Start", start_transitions)
    m.add_state("Python_state", python_state_transitions)
    m.add_state("is_state", is_state_transitions)
    m.add_state("not_state", not_state_transitions)
    m.add_state("neg_state", None, end_state=1)
    m.add_state("pos_state", None, end_state=1)
    m.add_state("error_state", None, end_state=1)
    m.set_start("Start")
    m.run("Python is great")
    m.run("Python is difficult")
    m.run("Perl is ugly")
```