Chapter 1

Playing with Texts in Python

The goal of this book is to teach you to how to write code to implement a solution you may propose in doing research on artificial intelligence in general and natural language processing specifically. But, before you start to write a program, you should know how to think like an AI scientist working on language related problems. For these problems, you need to learn to use formal concepts to denote linguistic and computational objects (specifically letters, words, texts, lexicons, and grammars). As a scientist, you should learn to formalize and represent the problem, and to use basic problem-solving strategies (divide-and-conquer, generate-and-test, state-space search, regression, neural networks). Only with an effective strategy, you can begin to write a program to solve the problem.

Most importantly, you will learn to observe, hypothesize, experiment, and evaluate as a scientist typically does.

More specifically, problem solving means the ability to formulate NLP problems, think creatively about computational solutions, and express a solution clearly and accurately. As it turns out, the process of learning to program in Python for NLP research is easy and a lot of fun. That's why this chapter is called, "Playing with Texts in Python."

If you are new to Python, you will be on a fast lane towad fluent Python. We will not walk

you through "Hello World!" and alike. If you already speak Python, you will surely be honing your skills. Python is a general purpose programming language, which is useful for almost anything computational. The past decades have witness the rise of the computational paradigm followed by the data-intensive paradigms. To be successful in research, you need to learn to use programming as a tool to conduct experiments and evaluation, often using very large scale datasets.

1.1 The Python programming language in a nut shell

Python is an example of a very high-level programming language. Although no two languages are complete alike, Lisp comes close to Python in many ways. The advantages of using Python or Lisp for AI research are obvious. First, it is much easier to program in such such very high level languages. With Python (or Lisp), you will be writing shorter programs that are easier to read, and take less time to write, and are more likely to be correct the first time (easier to debug). Therefore, more and more people are switching to Python to learn programming and to conduct research in all areas, from astronomy to artificial intelligence.

Python is distinctive in the following way:

Neat layout: The most striking feature of Python is its use of indentation and color (:) as both layout and program structure. Indentation for other languages are not mandatory and make programs look nicer to read. But, Indentation ia mandatory for Python program. There is no annoying *begins* and *ends*. Python use indentation to represent the structure and logical blocks of a program, whereas almost all other programming languages use keywords (e.g., **begin**, **end**) or symbols (e.g., {, }). A next structure starts right after a color must be indented (with 4 spaces suggested). Remember, when you switch from any language to Python, that will will forget to put a colon

hundred of times. However, you will get used to it before long.

Recursive data structure: Python is centered around the data type **list** (and the immutatable counterpart **tuple**). Lists in Python, similar to the **list** in Lisp, are dynamic, heterogeneous, recursively defined, with garbage collection. (cf. `https://norvig.com/python-lisp.html`. However, as you will find out soon, you can do a lot more with list. For example, instead of building up a list in a **for** loop, you can use **list comprehension** to do so.

In the most simple for, list comprehension simple let you apply (or **map**) an expression or a function to each item of a list to generate a more complex list. You can also add an **if-part** to a list comprehension to **filter** out item you do not want to keep.

Procedural paradigm Many programming languages are designed with an emphasis on one particular approach to programming. This often makes it difficult when the user need to write programs (occasionally) using a different approach. Python is a multi-paradigm programming languages (like Lisp and C++) that support several different approaches. In a large Python program, different parts might be written using different approaches

Like most programming languages (e.g., C and Pascal), Python has typical **procedural feature**s including variables, assignment, sequence, loop, conditional. However, much like SQL, Python program can be written to be non-procedural and very declarative. You write in a specification style describing the problem and solution and leave the interpreter to handle how to proceed with the steps of the solution efficiently.

Functional Programming Paradigm Python also support **functional programming** `https://docs.python.org/3/library/functional.html`) with **functional features** in modules like **itertools**, **funtools**, and **operators** that act as function of a function. Functional programming decomposes a problem into a set of functions that take

inputs and produce outputs without the side-effect of changing internal state (variables).

In a functional program, input flows through a set of functions. Each function operates on its input and produces some output. Functional style discourages functions with side effects that modify internal state or make other changes that aren't visible in the function's return value. Functions that have no side effects at all are called purely functional. Avoiding side effects means not using data structures that get updated as a program runs; every function's output must only depend on its input. Functional programming offers the advantages

- It is easier to prove a functional program works correctly.
- A functional program consists of a set of function is obviously more modular and easier to understand and modify.
- Functions can build on top of other function to compose more and more complex programs solving complex problems.
- Functional programs without internal state are easier to debug and test out.

Object-Oriented Programming Paradigms Although not strictly enforced, Python also supports **object-oriented programming** (OOP) with features like **class**, **inheritance**, and **polymorphism** (overloaded operators), and more. Object-oriented programs manipulate collections of objects. Objects have internal state and support methods that access or modify this internal state. OOP is the opposite of functional programming. Unlike pure object-oriented languages like Smalltalk and Java, Python support object-oriented programming, but does not force the user to use of object-oriented features.

1.2 Invoking Python in More than One Way

Python programs are handled by an interpreter directly without the hassle of generating machine code, linking, and memory allocation (things needed to be done in order to be executed under the operating system). You talk to the interpreter and things get done right away and right inside the interpreter, in a so-called **Read-Eval-Print Loop** (REPL). What is more. On most platforms, there are more than one way to invoke and run the Python interpreter:

1. **Interactive Mode** In this basic mode, you interact with Python one line at a time. After invoking Python from the command line, you then respond to the chevron-shaped prompt, and type a couple of lines of code. Python interpreter then reads the code, execute the code, and print out the results to he console. After that, Python prompts you again for the next line of code (REPL). The code can be a simple expression or a function call (will be automaticall printed without using a **print**() function. This is a great way to test out the built-in operators and functions (Section 1.x.x) you just switched from a different programming language and started to get familiar with Python.

```
$ Python
>>> '1 9 2 8 7 3 6 3 8 9 3 7'.split()
['1', '9', '2', '8', '7', '3', '6', '3', '8', '9', '3', '7']
>>> set('1 9 2 8 7 3 6 3 8 9 3 7'.split())
{'3', '8', '7', '1', '9', '6', '2'}
>>> a = set( '1 9 2 8 7 3 6 3 8 9 3 7'.split())
>>> sorted(a)
['1', '2', '3', '6', '7', '8', '9']
```

2. **Program Mode** Run Python to execute a program file, and examine the results printed on the console or put out to an newly created file. For example, you can use your favorite editor (TextMate, Sublime, or Pico) to create a file (say *test.py*) as follows:

```
a = '1 9 2 8 7 3 6 3 8 9 3 7'.split()
```

```
a = set(a)
print( sorted(a) )
```

After preparing the file **test.py**, you can then run Python to execute the program and get the result either ways (the console or an output file).

```
$ Python test.py
['1', '2', '3', '6', '7', '8', '9']
$ Python test.py > out.txt
$ cat out.txt
['1', '2', '3', '6', '7', ' 8', '9']
$
```

Keep in mind that, when in **program mode**, you will need to use the built-in function *print*() to actually print something. The REPL model does not work here.

3. Execute a short, in-line Python program Invoke Python with a in-line program, execute the program, and examine the results as they appear in the console or an newly created file.

```
$ Python -c "print(sorted(set('1 9 2 8 7 3 6 3 8 9 3 7'.split()))"                "
['1', '2', '3', '6', '7', '8', '9']
$
```

4. Program-then-Interactive Mode Invoke Python with a program, execute the program, and go on to interact with Python from the command line.

```
$ Python -i 2.py
>>> 'Colorless green ideas sleep furiously.'.split()
['Colorless', 'green', 'ideas', 'sleep', 'furiously.']

Same as using the built-in print function:
>>> print('Colorless green ideas sleep furiously.'.split())
['Colorless', 'green', 'ideas', 'sleep', 'furiously.']
```

5. Python CGI invoked by a Web server A CGI script is simply ... Simple Python programs (CGI scripts), based Common Gateway Interface (CGI) can be written to provide web services. The first line of the CGI script should be nonexecutable and point

to the location of the Python interpreter using the #!<path> syntax. When called by by the web server (or from a browser locally), the CGI is executed accordingly and all output to stdout is directed back to the web browser. So, you guessed it: all you simply print some HTML code to to **stdout** for the browser to process and displace on screen.

```
#!/usr/bin/python
webText = """
<H1>Useful Python Links</H1>
.. .
"""
print("Content-type: text/html\n")
print("<title>CGI Text</title>\n")
print(webText)
```

However, things could get unattainable when building a Web site with many interlinked pages in various directory. You will need to use a so-called framework such as Flask to get things done quicker. We will show how to do that in a later chapter.

1.3 Python Features at a Glance

Instructors and students in AI courses asked Peter Norvig to translate the Lisp code in the Russell & Norvig AI textbook into Java. They wanted Java because Java is a language familiar to them from other courses (that has been less and less so). Other reasons had to do with wanting to write graphical and Web applications for browsers.

Students also had a hard time getting used to Lisp syntax when they started with an AI class. However, Norvig found that rewriting AI programs in Java is not a good idea. Java rewrites were much longer and verbose, so they were strikingly different from the pseudocode in the book. Have looked around for a language, Norvig discovered Python was the closest to Lisp (and pseudocode) for a number of reasons:

AI-ready: Python is an excellent language for AI. Done properly, Python codes can look much more like typical pseudo-code found in AI textbooks than Lisp code does.

User-friendliness: Python is easy to use (interactive with no compile-link-load-run cycle), which is important for beginers and seasoned programmers alike.

Readabiliy: Python programs are definitely easier to understand than programs written in any other languages. This is true for a novice with limited or no experience in programming languages, and also for seasoned programmers. And readability is obviously very important in a pedagogical setting. Furthermore, remember you spend more time reading your own code (when debugging, revising, and expanding the program) than the time you spend in writing!

I will go through the basic features of Python:

Atomic data type: These are the basic data with different mathematical property, including integer (regular round numbers or big numbers), float (real numbers), character (in roman alphabet or non-Western language like Chinese), Boolean (True or False), and missing or undefined value (None which arises when you forget return a value from a function). Basic data are supported with common infix operators including +, -. *, /. Boolean values are typically associated with the conditional statement (if and while statements) to determine the flow of program execution.

Basic Data Types	**Python Data Types**
Integer	42
Bignum	100000000000000000
Float	12.34
Character	'a', \n, \t
Boolean	True, False
Falsehood	False, None, 0, '', [] , {}
Truth	Anything else
Conditional value	y if x else z
Missing value	None

Compound data: Compound data are typically sequences of atomic data. Strings are an ordered sequence of characters, Lists (or tuples) are an ordered sequence of mixed bag of atomic or compound data items. Set is a collection of distinct and un-ordered items. Dictionary (or hashtable) is a set of unique key and value supporting

Compound Data	**Python Data Types**
String	"hello" *or* 'hello' (**immutable)**
multiply	'=' * 10
List	
literal	[1, 2.0, "three"], [] (empty list)
append	x + [y] == x.append(y)
access	x [0], x[3], x[i], x[-1]
slice	x [2:3], x[start:end], x[start:], x[:end]
equality	x == y
same	x is y
size	len(seq)
multiply	[[] for i in range(10)]
Tuple	(1, (2.0, ("three", None))), (,) (empty tuple)
Set	{ 3, 1, 2 } (no fixed order)
Hashtable	
initialize	h = {}
populate	h ["one"] = 1.0
access	h ["one"] *or* h.get("one")
literal	h = {"one": 1, "two": 2}
grammar as a hashtable	{ 'S' : 'NP VP', 'NP': 'Art N', 'VP' : 'V NP', 'Art' : 'the | a', 'N' = 'man | ball | woman', 'V' : 'hit | took | saw | liked' }
Class (object)	class Stack: ...
Instance	Stack()
Stream	open("file")

Control: Python provides typical control structure of typical procedural languages, including if-statements, loops, function calls, and sequences of statements (assignments or nested control statements). Indentation is main program layout and structure mechanism. This striking design make Python code more concise. Gone are the begins, the ends, the curly brackets. There is need for using 'begin' after 'begin' and then 'end' after 'end' (or curly brackets). You simply indent the line to begin a new block and un-indent to finish the block.

Built-in functions: As part of Python language (no need to import from a module), com-

Control Structure	Examples
Function	
named	>>> def transpose (m): >>> return zip(*m)
function call	>>> transpose([[1,2,3], [4,5,6]]) `[(1, 4), (2, 5), (3, 6)]`
unamed	lambda x: x + x

Conditional statement	if <boolean>:
	<statement>
	elif:
	<statement>
	else:
	<statement>

Loop	while <boolean>: <statement>
sequence in loop	<statement> for <list> in <iterator>: (or for <var> in range(n):) <statement> <statement>
Assignment	x = <expression> x, y = 1, 2 x, y = y, x
unpack a list	x, y, z = w x.slot = y
Exceptions	assert denom != 0, "denom != 0" try: attempt() finally: recovery() try: ...; raise 'ball' except 'ball': ...
Other	with ___ : statement

mon functions are built in so the user can concisely manipulate the data (so write **abs(i)** for short, instead of **-i if i < 0 else i**). These include abs() for converting to an absolute value, round() for rounding a float to an integer, len() for the length of a string or list.The range() function generate a list of integer, typically used to control a for loop. Alternatively, you can use enumerate() to iterate through the items of a list with access to the index as you move along. You can also conveniently take the maximal, minimal values of a list using max() and min(). When the item is a list, max() and min() return the argument that optimize a certain field of the items.

Built-in functions	Python Data Types
Integer, float	abs(-1) >>> 1, abs(12.3) >>> 12.3
Float	round(1.3) >>> 1, round(1.6) >>> 2, floor(1.3) >>> 1,
String	len('aaaa') >>> 4, len([1, 2, 3]) >>> 3,
List, tuple	range(3) >>> [0, 1, 2] enumerate(['a', 'b', 'c']) >>> [(0, 'a', (1, 'b'), (2, 'c')] min((1, 2.0, 3)) >>> 1 max((1, 2.0, 3)) >>> 3 sum((1, 2.0, 3)) >>> 6.0 list(zip((1, 2, 3), 'abc')) >>> [(1, 'a'), (2, 'b'), (3, 'c')] next(), len()

List & function	>>> list(map(xxxx, (1, 2.0, 3)))
	yyyy
	>>> sorted((1, 2.0, 3), key=lambda x: x[0])
	yyyy
	reduce(operator.add, numbers)
	all(x%2 for x in [1,3,5]) = > True
	any(x%2 for x in [1,2,3]) = > True
	filter(lambda x: x%2 == 0, numbers)
	or [x for x in numbers if x%2 == 0]
	min(numbers)

Stream	print(), open(), input()
Cast	int(), float(), list(), format(), str(), ascii()
Optional arg	
Variable-length arg	def f (*arg): ...
Unspecified keyword args	def f (**arg): ...
function call with keywords	def f (arg=val): ... f(y=1, x=2)

doc strings	def f(x):
	"compute f value"
	>>> f.__doc__
	"compute f value"

1.3.1 Python Reserved Words

A limited set of words are reserved for basic operations in Python, including

(1) Boolearn expression: **and or not is**

(2) Control structures: **while if else elif break for continue with in**

(3) Subroutine and class: **def return lambda yield pass class**

(4) Exception: **except raise finally try**

(5) Module: **import as from**

(6) Other: **del global print assert exec with**

You can print them out for different versions of Python:

```
>>> import keyword
>>> keyword.iskeyword('break')
True
>>> keyword.kwlist
['False', 'None', 'True', 'and', 'as', 'assert', 'break', 'class', 'continue', 'def',
 'del', 'elif', 'else', 'except', 'finally', 'for', 'from', 'global', 'if', 'import',
 'in', 'is', 'lambda', 'nonlocal', 'not', 'or', 'pass', 'raise', 'return', 'try',
 'while', 'with', 'yield']
```

Certain classes of identifiers (besides keywords) have special meanings. These identifiers (*dunder variables)* are identified by the patterns of leading and trailing underscore characters (e.g., __doc__).

1.4 Python by Example–as easy as one two three

A fun way to learn Python is starting from one-line program and gradually move on to longer and longer programs: two lines, three line programs, so on and so forth. Before you know it, you learned a lot.

1.4.1 One-liners

You can write a one-line code to turning a sentence into a list of words (from string to list)

```
$ Python
>>> 'Colorless green ideas sleep furiously.'.split()
['Colorless', 'green', 'ideas', 'sleep', 'furiously.']

Same as using the built-in print function:
>>> print('Colorless green ideas sleep furiously.'.split())
['Colorless', 'green', 'ideas', 'sleep', 'furiously.']
```

1.4.2 Two-liners

In just two lines, you can write a function that turns a sentence into a list of word tokens, treating punctuation marks as token.

```
>>> import re
>>> def words(text): return re.findall(r'([a-z]+|[.,:?!])', text.lower())
```

```
Run the program:
>>> words('Colorless green ideas sleep furiously.')
['colorless', 'green', 'ideas', 'sleep', 'furiously', '.']

Or

$ echo -e "import re\ndef words(text): return re.findall(r'([a-z]+|[.,:?!])',
text.lower())\nprint(words('Colorless green ideas sleep furiously.'))" > my.py
$ python -i my.py
>>> words('Colorless green ideas sleep furiously.')
['colorless', 'green', 'ideas', 'sleep', 'furiously', '.']
```

Put the list back to a string:

```
>>> ' '.join(words('Colorless green ideas sleep furiously.'))
'colorless green ideas sleep furiously .'

Squeeze out the space:

>>> ''.join(words('Colorless green ideas sleep furiously.'))
'colorlessgreenideassleepfuriously.'
```

Another two-line program executed from the command line:

```
$ python -c 'import time; time.sleep(60)' | pv
0.00 B 0:00:01 [0.00 B/s] [<=>               ] -- Started
0.00 B 0:01:00 [0.00 B/s] [<=>               ] -- Ended
```

1.4.3 Three-liner

Return a list of all possible (first, rem) pairs, len(first)<=L

```
>>> def splits(text, L=10):
>>>    return [(text[:i+1], text[i+1:])
>>>            for i in range(min(len(text), L))]

Run the function:
$ python -i my.py
>>> from pprint import pprint
>>> pprint(splits('colorlessgreenideassleepfuriously.'))
[('c', 'olorlessgreenideassleepfuriously.'),
 ('co', 'lorlessgreenideassleepfuriously.'),
 ('col', 'orlessgreenideassleepfuriously.'),
 ('colo', 'rlessgreenideassleepfuriously.'),
 ('color', 'lessgreenideassleepfuriously.'),
 ('colorl', 'essgreenideassleepfuriously.'),
 ('colorle', 'ssgreenideassleepfuriously.'),
```

```
 ('colorles', 'sgreenideassleepfuriously.'),
 ('colorless', 'greenideassleepfuriously.'),
 ('colorlessg', 'reenideassleepfuriously.')]
>>>
```

1.4.4 Four-liner

Estimate the probability of a word

```
N = 1024908267229 ## Size of Google Web 1T Dataset
word_count = [ line.split('\t') for line in open('count_1w.txt', 'r') ]
Pdist = dict( [ (word, float(count)/N) for word, count in word_count ] )

def Pw(word): return Pdist[word] if word in Pdist else 10./10**len(word)/N

Run the function:
>>> pprint [ (w, Pw(w)) for w in words('Colorless green ideas sleep furiously.') ]
[('colorless', 5.0e-07),
 ('green', 0.00011),
 ('ideas', 6.6e-05),
 ('sleep', 2.9e-05),
 ('furiously', 4.4e-07)
 ('.', 9.76e-13) ]
>>> print( map(Pw, words('Colorless green ideas sleep furiously.') ))
[ 5.0e-07, 0.00011, 6.6e-05, 2.9e-05, 4.4e-07, 9.76e-13 ]
>>>
```

Let us look at a completely different problem: computing Fibonacci numbers (as shown below).

```
def fib(n):
    if n == 0:
        return 0
    elif n == 1:
        return 1
    else:
        return fib(n-1) + fib(n-2)

$ python -i fib.py
>>> fib(40)
102334155
>>> from timeit import timeit
>>> timeit.timeit("fib(15)", setup="from __main__ import test")
246 (sec.)
```

The recursion function **fib()** run top-down, and gradually aggregate the partial solutions from the ground up to yield the final result.This process runs too slowly, because $fib(i)$ is

recalcuated for $i < 40$ over and over again. Solution: memorizing the values rather than redoing the calculation.

Specifically, we can write a **memorize** function which turns **fib()** into the **helper()** function, which calls fib() to solve all the sub-problems (and memorize the results) when it encounter them the first time. It then return the results by looking up the functional values from the dictionary thereafter. So, that ensures any function call is only evaluated once.

```
def memoize(f):
    memo = {}
    def helper(x):
        if x not in memo:
            memo[x] = f(x)
        return memo[x]
    return helper

fib = memoize(fib)

print(fib(40))
```

Python provide decoration (with the @ notation) for this. Decoration applies the class to the function to augment the function with a dictionary, making it more efficient.

```
class memoize:
    def __init__(self, fn):
        self.function = fn
        self.memodict = {}

    def __call__(self, *args):
        if args not in self.memodict:
            self.memodict[args] = self.function(*args)
        return self.memodict[args]

@memoize
def fib(n):
    if n == 0:
        return 0
    elif n == 1:
        return 1
    else:
        return fib(n-1) + fib(n-2)
```

```
def fib_no_memo(n):
    if n == 0:
        return 0
    elif n == 1:
        return 1
    else:
        return fib(n-1) + fib(n-2)

$ python -i myfib.py
>>> fib(40)
102334155
```

1.4.5 Five lines

Return a sentence with the best segmentation of the input text without spaces

```
@memoize
def segment(text):
    if not text: return []
    candidates = ([first]+segment(rem) for first,rem in splits(text))
    return max(candidates, key=lambda: x product(P(w) for w in x))

Run the function:
>>> print(segment('colorlessgreenideassleepfuriously.'))
['colorless', 'green', 'ideas', 'sleep', 'furiously', '.']
>>> print(' '.join(segment('colorlessgreenideassleepfuriously.')))
'colorless green ideas sleep furiously .'
```

1.4.6 Six lines

Read word counts to estimate word probability

```
import re, collections

def words(text):
    return re.findall(r'\w+', text.lower())

word_count = collections.Counter(words(open('big.txt').read()))

def P(word, N = sum(word_count.values())):
    return word_count[word]/N

$ python -i 6.py
>>> pprint( map(P, words('speling spelling speeling')))
[('speling', 0.0), ('spelling', 3.59e-06), ('speeling', 0.0)]
```

1.4.7 Seven lines

Produce words within one edit of the input

```
letters    = 'abcdefghijklmnopqrstuvwxyz'

def edits1(word):
    splits     = [(word[:i], word[i:])    for i in range(len(word) + 1)]
    deletes    = [L + R[1:]               for L, R in splits if R]
    transposes = [L + R[1] + R[0] + R[2:] for L, R in splits if len(R)>1]
    replaces   = [L + c + R[1:]           for L, R in splits if R for c in letters]
    inserts    = [L + c + R               for L, R in splits for c in letters]
    return set(deletes + transposes + replaces + inserts)
```

```
$ python -i 6.py
>>> word = 'speling'
>>> pprint([(word[:i], word[i:])    for i in range(len(word) + 1)])
[('', 'speling'),
 ('s', 'peling'),
 ('sp', 'eling'),
 ('spe', 'ling'),
 ('spel', 'ing'),
 ('speli', 'ng'),
 ('spelin', 'g'),
 ('speling', '')]
>>> pprint(  [(L, c, R) for L, R in splits for c in 'l'] )
[('', 'l', 'speling'),
 ('s', 'l', 'peling'),
 ('sp', 'l', 'eling'),
 ('spe', 'l', 'ling'),
 ('spel', 'l', 'ing'),
 ('speli', 'l', 'ng'),
 ('spelin', 'l', 'g'),
 ('speling', 'l', '')]
>>> pprint(  [L + c + R  for L, R in splits for c in 'l'] )
['lspeling',
 'slpeling',
 'spleling',
 'spelling',
 'spelling',
 'spelilng',
 'spelinlg',
 'spelingl']

>>> pprint( list(edits1('speling')) )
['spelinx', 'spebling', 'spelinf' ... ]
>>> pprint( list(map(lambda x: (x, P(x)), list(edits1('speling')))) )
[('spjling', 0.0),
 ('bspeling', 0.0),
```

```
 ('spelint', 0.0), ...
 ('spelling', 3.5e-6), ...

>>> print( list(filter(lambda x: P(x) != 0.0, edits1('speling'))) )
['spelling']
>>> print( max(edits1('speling'), key=P) )
spelling
```

1.4.8 Eight lines

Peter Norvig famously wrote a 21-line Python program (How to Write a Spelling Corrector, `http://norvig.com/spell-correct.html`), Norvig's program later became a Kaggle task (`https://www.kaggle.com/bittlingmayer/spelling`)

To do it quick and dirty, Norvig resort to brute-force generate-and-test strategy ():

```
def correction(WORD):
    if P(WORD) > 0: return WORD
    Generate candidates C1 with one WORD away from word
    If there exists a candidate x in C1, P(x) > 0:
        return argmax(x) P(x) for x in C1
    Generate candidates C2 with one edit away from any c in C1
    If there exists a candidate x in C2, P(x) > 0:
        return argmax P(x) for x in C2
```

This pseudocode is implemented with elegant Short-circuit evaluation, a.k.a, minimal evaluation or McCarthy evaluation (named after one of funding fathers of AI, John McCarthy). See for more details. When evaluating a Boolean expression (e.g., A or B or C) up to a certain point (e.g., A==False and B==True, Python will bypass the rest of the expression, if the value can be determined (i.e., A or B or C== True) regardless of the value of the rest of the expression. A case in point: *known([word]) or known(edits1(word)) or known(edits2(word)) or [word]* in the candidates() function shown below:

```
def correction(word):
    return max(candidates(word), key=P)

def candidates(word):
    return (known([word]) or known(edits1(word)) or known(edits2(word)) or [word])
```

```
def known(words):
    return set(w for w in words if w in WORDS)

def edits2(word):
    return (e2 for e1 in edits1(word) for e2 in edits1(e1))

$ python -i 8.py
>>> print('speling -->', correction('speling'))
speling --> spelling
```

1.4.9 Nine lines

Perform unit test.

```
def unit_tests():
    assert correction('speling') == 'spelling'              # insert
    assert correction('korrectud') == 'corrected'           # replace 2
    assert Counter(words('This is a test. 123; A TEST this is.')) == (
           Counter({'123': 1, 'a': 2, 'is': 2, 'test': 2, 'this': 2}))
    assert P('quintessential') == 0
    assert 0.07 < P('the') < 0.08
    return 'unit_tests pass'

>>> ...
```

1.4.10 Ten lines

Produce words within one edit of the input

```
def spelltest(tests): # Run correction(wrong) on all (right, wrong) pairs
    good, unknown = 0, 0
    for right, wrong in tests:
        w = correction(wrong)
        if w == right: good += 1
        else: unknown += (right not in WORDS)
    n = len(tests)
    print('{:.0%} of {} correct ({:.0%} unknown) '.format(good / n, n, unknown / n))

if __name__ == '__main__':
    spelltest(Testset(open('spell-testset1.txt')))

$ python -i 10.py
>>> spelltest(Testset(open('spell-testset1.txt')))
...
...
```

1.4.11 Twelve lines

Produce words within one edit of the input

```
from random import choice

grammar = dict( S = [['NP','VP']], NP = [['Art', 'N']], VP = [['V', 'NP']],
        Art = ['the', 'a'], N = ['man', 'ball', 'woman', 'table'], V = ['hit', 'took', 'saw', 'liked'

def generate(phrase): # Generate a random sentence or phrase
    if isinstance(phrase, list): return mappend(generate, phrase)
    elif phrase in grammar:      return generate(choice(grammar[phrase]))
    else:                        return [phrase]

def generate_tree(phrase): # Generate a random sentence or phrase, with a complete parse tree
    if isinstance(phrase, list): return map(generate_tree, phrase)
    elif phrase in grammar:      return [phrase]+generate_tree(choice(grammar[phrase]))
    else:                        return [phrase]

def mappend(fn, args): # Map function over the args; append results.
    return [item for result in map(fn, args) for item in result]

$ python -i 12.py
>>> generate('S')
['the', 'man', 'saw', 'the', 'table']

>>> ' '.join(generate('S'))
'the man saw the table'
>>>
```

Note: This is adopted from a Python program translated from Lisp as an example in the widely adopted AI textbook (*Artificial Intelligence: A Modern Approach*). See Norvig's post, *Python for Lisp Programmers* (`https://norvig.com/python-lisp.html#sample`). for more details.

Alternatively, you can store code in a file and use the interpreter to execute the contents of the file, which is called a **script**. By convention, Python scripts have names that end with `.py`.

To execute the script, you have to tell the interpreter the name of the file. If you have a script named `dinsdale.py` and you are working in a UNIX command window, you type

`python dinsdale.py`. In other development environments, the details of executing scripts are different.

Working in interactive mode is convenient for testing small pieces of code because you can type and execute them immediately. But for anything more than a few lines, you should save your code as a script so you can modify and execute it in the future.

1.5 Exercise: Spelling check based on beam search

You are asked to write a shorter, more readable, and more efficient version of Peter Norvig's `spell.py`. For that, you are going to start with the initial state (having done nothing), (L, R), $L = ''$ and $R = W$. Each move from one state to another represent handle the first character of R shortening R in the process. Eventually, the search stop, when reaching states with nothing left to be handle (i.e., state = $(W', '')$. The solution is among such W' with high probability $P(W)$ and the minimal number of edits (at most 2 edits). The pseudocode of state-space search is the following:

```
def correction(word):
"State-space search"
    STATES = {initial_state}
    For each character R0 in word (word = L + R0 + R1:
    Generate next states N for each state in STATES
    Combine the next states N
     return argmax(x) P(x) for x in STATES

def correction(word):
    states = [ (_____, _____, _____, _____) ]
    for i in range(len(word)):
        states = [ state for states in ____________________ for state in states ]
        ____________________
        ____________________
        states = sorted(states, key=_____) [:MAXBEAM]
    return max(states, key=lambda x: ____________________)
```

To implement the pseudocode, you need to do the following :

1. Word probability: Reuse the 4-line program for segment() (word segmentation).

```
N = 1024908267229 ## Size of Google Web 1T Dataset
word_count = [ line.split('\t') for line in open('count_1w.txt', 'r') ]
Pdist = dict( [ (word, float(count)/N) for word, count in word_count ] )

def Pw(word): return Pdist[word] if word in Pdist else 10./10**len(word)/N
```

2. Represent the states: For the given word *W*, the partial solution is represented by the so-called "state," (*L*, *R*, *ED*, *P*) where *L* is the edited version of *W*-*R*, *R* is the part is yet to be edited, *ED* is the part is yet to be edited, and *P* is the part is yet to be edited.

3. Write a next-state function: Write a function that , given a state, generate a list of next states based on the edit1() function.

```
def edits1(word):
    "All edits that are one edit away from `word`."
    letters    = 'abcdefghijklmnopqrstuvwxyz'
    splits     = [(word[:i], word[i:])    for i in range(len(word) + 1)]
    deletes    = [L + R[1:]               for L, R in splits if R]
    transposes = [L + R[1] + R[0] + R[2:] for L, R in splits if len(R)>1]
    replaces   = [L + c + R[1:]           for L, R in splits if R for c in letters]
    inserts    = [L + c + R               for L, R in splits for c in letters]
    return set(deletes + transposes + replaces + inserts)

letters    = 'abcdefghijklmnopqrstuvwxyz'
def next_states(state):
    L, R, edit, prob = state
    R0, R1 = R[0], R[1:]
    if edit == 2: _____________
    noedit = ___________________
    delete = ___________________
    replace = __________________
    insert  = __________________
    return ____________________

$ python -i spell.prob.py
>>> word =  'appearant'
>>> pprint(next_states( ('', word, 0, P[word]) ))
[('a', 'ppearant', 0, 0.0),
 ('', 'ppearant', 1, 0.0),
 ('a', 'ppearant', 1, 0.0),
 ('b', 'ppearant', 1, 0.0),
...
 ('z', 'ppearant', 1, 0.0),
 ('aa', 'ppearant', 1, 0.0),
 ('ab', 'ppearant', 1, 0.0),
...
```

```
 ('az', 'ppearant', 1, 0.0)]
>>>
```

4. Beam search: Write a function that perform state space search with simple pruning:

```
def correction(word):
    states = [ <initial state> ]
    for i in range(len(word)):
        print(i, states[:3])
        STATES = < map the next_states function to each state in STATES
         to get a list (of lists) of states>
        <Combine states with the same L and R so the #edit is minimized>
        <Sort the states in STATES in increasing values of #edit first and
         then decreasing probability of state>
        <Prune STATES, leaving at most MAXBEAM number of states>
    <Filter STATES and keep only states with #edit==0 or probability>0>
    <If there are some plausible edited results (#edit>0 and probability>0
     then remove the state with #edit==0>
    return <the first three of plausible corrections sorted by probability>
```

```
$ python -i spell.prob.py
>>> word =  'appearant'
>>> correction(word')
correction('appearant')
0 [('', 'appearant', 0, 0.0)]
1 [('a', 'ppearant', 0, 0.0), ('', 'ppearant', 1, 0.0), ('aa', 'ppearant', 1, 0.0)]
2 [('ap', 'pearant', 0, 0.0), ('a', 'pearant', 1, 0.0), ('aa', 'pearant', 1, 0.0)]
3 [('app', 'earant', 0, 0.0), ('aap', 'earant', 1, 0.0), ('aapp', 'earant', 1, 0.0)]
4 [('appe', 'arant', 0, 0.0), ('aape', 'arant', 1, 0.0), ('aappe', 'arant', 1, 0.0)]
5 [('appea', 'rant', 0, 0.0), ('aapea', 'rant', 1, 0.0), ('aappea', 'rant', 1, 0.0)]
6 [('appear', 'ant', 0, 0.0), ('aapear', 'ant', 1, 0.0), ('aappear', 'ant', 1, 0.0)]
7 [('appeara', 'nt', 0, 0.0), ('appare', 'nt', 2, 4.2e-05), ('aapeara', 'nt', 1, 0.0)]
8 [('appearan', 't', 0, 0.0), ('apparen', 't', 2, 4.2e-05), ('aapearan', 't', 1, 0.0)]

  [('appearance', '', 2, 0.00012,
    ('apparent', '', 2, 4.2e-05),
    ('appearing', '', 2, 2.1e-05)]
>>>
```

References

Harshit Pande. 2017 (ACL). Effective search space reduction for spell correction using character neural embeddings http://www.aclweb.org/anthology/E17-2027

Yanen Li, Huizhong Duan and ChengXiang Zhai. 2012 (SIGIR). A Generalized Hidden Markov Model with Discriminative Training for Query Spelling Correction http:

//sifaka.cs.uiuc.edu/czhai/pub/sigir12-spelling.pdf

David Currie. 2017. Creating a Spell Checker with TensorFlow https://towardsdatascience.com/creating-a-spell-checker-with-tensorflow-d35b23939f60