Fundamentals of Programming I
Strings

Sequences

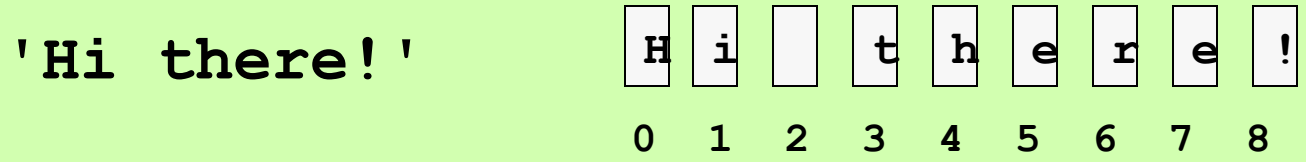
- *Sequences* are collections of data values that are ordered by position
- A *string* is a sequence of characters
- A *list* is a sequence of any Python data values
- A *tuple* is like a list but cannot be modified

String Assignment, Concatenation, and Comparisons

```
a = 'apple'
b = 'banana'
print(a + b)          # Displays applebanana
print(a == b)         # Displays False
print(a < b)          # Displays True
```

Strings can be ordered like they are in a dictionary

Positions or Indexes



Each character in a string has a unique position called its *index*

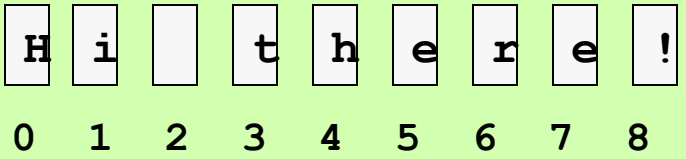
We count indexes from 0 to the length of the string minus 1

A **`for`** loop automatically visits each character in the string, from beginning to end

```
for ch in 'Hi there!': print(ch)
```

Traversing with a **`for`** Loop

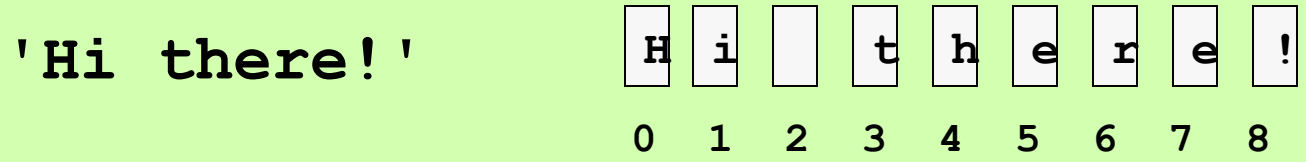
`'Hi there!'`



A **`for`** loop automatically visits each character in the string, from beginning to end

```
for ch in 'Hi there!': print(ch, end = '')


# Prints Hi there!
```

The Subscript Operator



Alternatively, any character can be accessed using the *subscript operator* **`[]`**

This operator expects an **`int`** from 0 to the length of the string minus 1

Example:

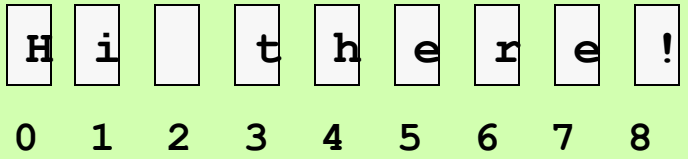
```
'Hi there!'[0]   # equals 'H'
```

Syntax:

```
<a string>[<an int>]
```

The **`len`** Function

`'Hi there!'`

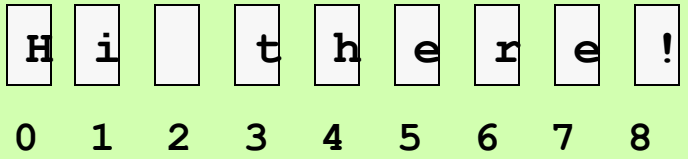


The **`len`** function returns the length of any sequence

```
>>> len('Hi there!')
9

>>> s = 'Hi there!'

>>> s[len(s) - 1]
'!'
```

An Index-Based Loop

'Hi there!'

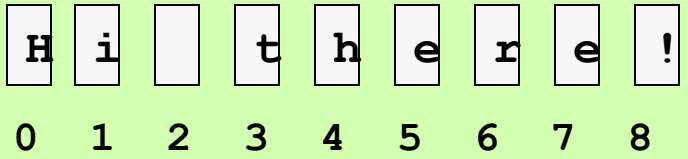


If you need the positions during a loop, use the subscript operator

```
s = 'Hi there!'

for ch in s: print(ch)

for i in range(len(s)): print(i, s[i])
```

Oddball Indexes

'Hi there!'

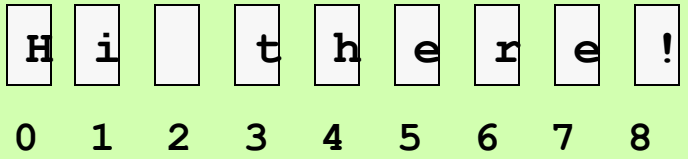


To get to the last character in a string:

```
s = 'Hi there!'

print(s[len(s) - 1])      # Displays !
```

Oddball Indexes

'Hi there!'



To get to the last character in a string:

```
s = 'Hi there!'

print(s[len(s) - 1])

# or, believe it or not,

print(s[-1])
```

A negative index counts backward from the last position in a sequence

Slicing Strings

Extract a portion of a string (a substring)

```
s = 'Hi there!'

print(s[0:])          # Displays Hi there!

print(s[1:])          # Displays i there!

print(s[:2])          # Displays Hi (two characters)

print(s[0:2])         # Displays Hi (two characters)
```

The number to the right of : equals one plus the index of the last character in the substring

String Methods

```
s = 'Hi there!'

print(s.find('there'))          # Displays 3

print(s.upper())                # Displays HI THERE!

print(s.replace('e', 'a'))      # Displays Hi thara!

print(s.split())                # Displays ['Hi', 'there!']
```

A *method* is like a function, but the syntax for its use is different:

```
<a string>.<method name>(<any arguments>)
```

String Methods

```
s = 'Hi there!'

print(s.split())                    # Displays ['Hi', 'there!']
```

A sequence of items in **[ ]** is a Python *list*

Characters in Computer Memory

- Each character translates to a unique integer called its *ASCII value* (American Standard for Information Interchange)

- Basic ASCII ranges from 0 to 127, for 128 keyboard characters and some control keys

The Basic ASCII Character Set

	0	1	2	3	4	5	6	7	8	9
0	NUL	SOH	STX	ETX	EOT	ENQ	ACK	BEL	BS	HT
1	LF	VT	FF	CR	SO	SI	DLE	DC1	DC2	DC3
2	DC4	NAK	SYN	ETB	CAN	EM	SUB	ESC	FS	GS
3	RS	US	SP	!	"	#	$	%	&	`
4	(	)	*	+	,	-	.	/	0	1
5	2	3	4	5	6	7	8	9	:	;
6	<	=	>	?	@	A	B	C	D	E
7	F	G	H	I	J	K	L	M	N	O
8	P	Q	R	S	T	U	V	W	X	Y
9	Z	[	\	]	^	_	'	a	b	c
10	d	e	f	g	h	i	j	k	l	m
11	n	o	p	q	r	s	t	u	v	w
12	x	y	z	{	|	}	~	DEL		

The **ord** and **chr** Functions

ord converts a single-character string to its ASCII value

chr converts an ASCII value to a single-character string

```
print(ord('A'))                    # Displays 65

print(chr(65))                     # Displays A

for ascii in range(128):           # Display 'em all
    print(ascii, chr(ascii))
```

Data Encryption

A really simple (and quite lame) encryption algorithm replaces each character with its ASCII value and a space

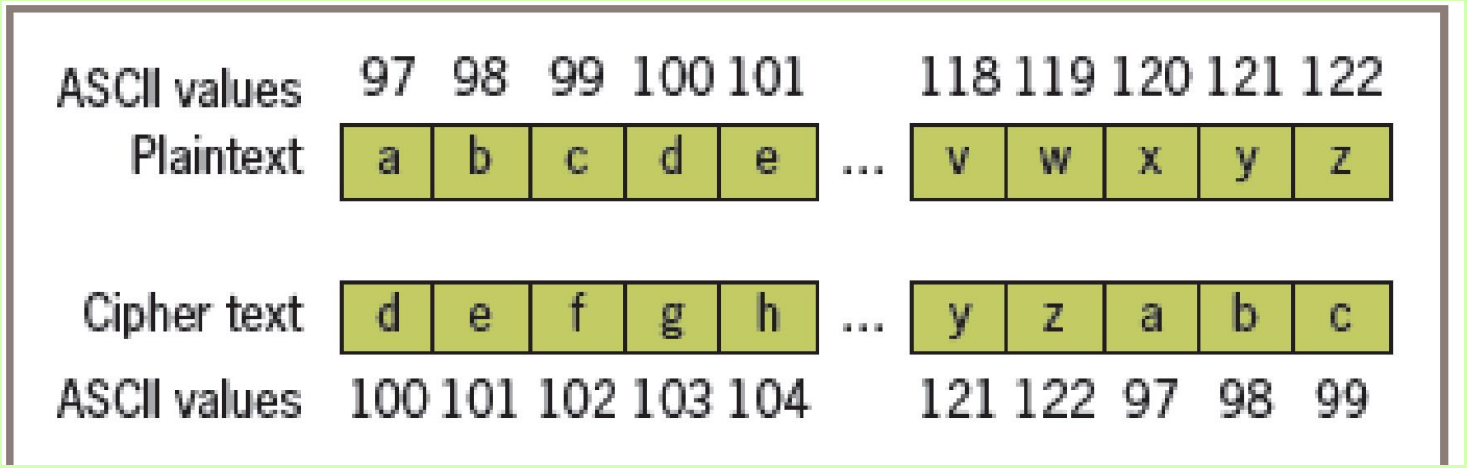
```
source = "I won't be here!"

code = ""
for ch in source:
    code = code + str(ord(ch)) + " "
print(code)

# Displays 73 32 119 111 110 39 116 32 98 101 32 104 101 33
```

Data Encryption

- The sender **encrypts** a message by translating it to a secret code, called a **cipher text**.
- At the other end, the receiver **decrypts** the cipher text back to its original **plaintext** form.
- Both parties to this transaction must have at their disposal one or more **keys** that allow them to encrypt and decrypt messages

- A very simple encryption method that has been in use for thousands of years is called a **Caesar cipher**.
- This encryption strategy replaces each character in the plaintext with the character that occurs a given distance away in the sequence.

- The string "invaders" would be encrypted as "lqydghuv“
- To decrypt this cipher text back to plaintext, you apply a method that uses the same distance value but looks to the left of each character for its replacement


ASCII values
97 98 99 100 101 118 119 120 121 122
Plaintext
a b c d e ... v w x y z
Cipher text
d e f g h ... y z a b c
ASCII values
100 101 102 103 104 121 122 97 98 99

Encryption

```
plainText = input("Enter a one-word, lowercase message: ")
distance = int(input("Enter the distance value: "))
code = ""
for ch in plainText:
    ordvalue = ord(ch)
    cipherValue = ordvalue + distance
    if cipherValue > ord('z'):
        cipherValue = ord('a') + distance - \
                      (ord('z') - ordvalue + 1)
    code += chr(cipherValue)
print(code)
```

Decryption

```
Decrypts an input string of lowercase letters and prints
the result. The other input is the distance value.
"""

code = input("Enter the coded text: ")
distance = int(input("Enter the distance value: "))
plainText = ""
for ch in code:
    ordvalue = ord(ch)
    cipherValue = ordvalue - distance
    if cipherValue < ord('a'):
        cipherValue = ord('z') - \
                      (distance - (ord('a') - ordvalue - 1))
    plainText += chr(cipherValue)
print(plainText)
```

Here are some executions of the two scripts in the IDLE shell:

```
Enter a one-word, lowercase message: invaders
Enter the distance value: 3
Lqydghuv

Enter the coded text: lqydghuv
Enter the distance value: 3
invaders
```

Decimal and Binary

- Decimal numbers use the 10 decimal digits and a base of 10
- Binary numbers use the binary digits 0 and 1 and a base of 2
- The base is often provided as a subscript to indicate the type of system, as in 3042_{10} and 11011110_2
- Thus, 1101_{10} represents a very different integer value from 1101_2

Positional Notation

- Number systems are *positional*, so the magnitude of the number depends on the base and the position of the digits in the number
- Each position represents a power of the number's base
- For example, in a 3-digit decimal number, the three positions represent the number of hundreds (10^2), tens (10^1), and ones (10^0)
- $342 = 3 * 10^2 + 4 * 10^1 + 2 * 10^0$
- $= 3 * 100 + 4 * 10 + 2 * 1$
- $= 300 + 40 + 2$
- $= 342$

Positional Notation: Binary

- The base is now 2 and the only digits are 0 and 1
- Each position represents a power of 2
- For example, in a 4-digit binary number, the four positions represent the number of eights (2^3), fours (2^2), twos (2^1), and ones (1^0)
- $1101 = 1 * 2^3 + 1 * 2^2 + 0 * 2^1 + 1 * 2^0$
- $= 1 * 8 + 1 * 4 + 0 * 2 + 1 * 1$
- $= 8 + 4 + 0 + 1$
- $= 13$

An Algorithm for Binary to Decimal Conversion

```
# Input: A string of 1 or more binary digits
# Output: The integer represented by the string
binary = input("Enter a binary number: ")
decimal = 0
exponent = len(binary) - 1
for digit in binary:
    decimal = decimal + int(digit) * 2 ** exponent
    exponent = exponent - 1
print("The integer value is", decimal)
```

The **`len`** function returns the number of characters in a string

The **`for`** loop visits each character in a string

Counting in Binary

Binary	Magnitude	
0	0	
1	1	
10	2	2^1
11	3	
100	4	2^2
101	5	
110	6	
111	7	
1000	8	2^3

Each power of 2 in binary is a 1 followed by the number of 0s equal to the exponent

What is the magnitude of 1000000_2?

Counting in Binary

Binary	Magnitude	
0	0	
1	1	$2^1 - 1$
10	2	
11	3	$2^2 - 1$
100	4	
101	5	
110	6	
111	7	$2^3 - 1$
1000	8	

Each number with only 1s equals one less than the power of 2 whose exponent is that number of 1s

What is the magnitude of 1111_2?

Convert Decimal to Binary

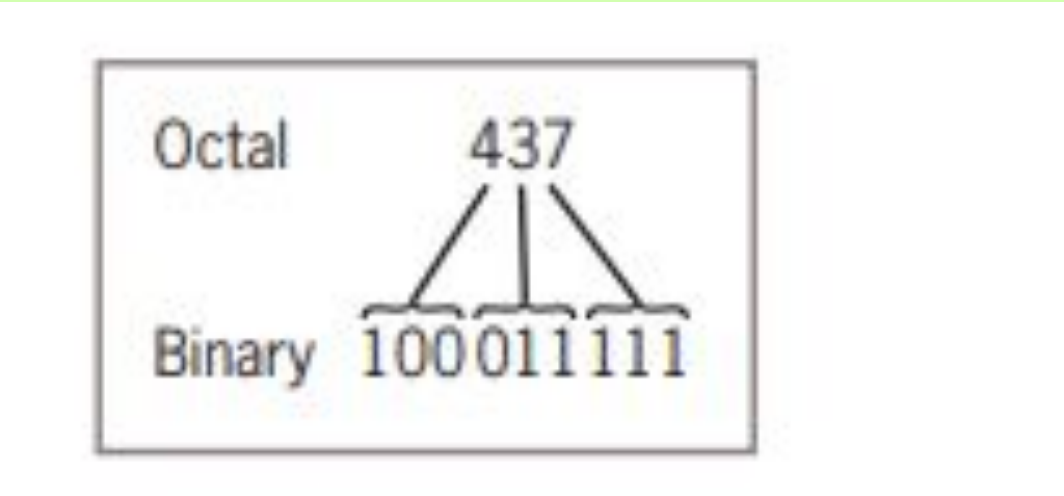
- Start with an integer, N, and an empty string, S
- Assume that N > 0
- While N > 0:
  - Compute the remainder of dividing N by 2 (will be 0 or 1)
  - Prepend the remainder's digit to S
  - Reset N to the quotient of N and 2

An Algorithm for Decimal to Binary Conversion

```
# Input: An integer > 0
# Output: A string representing the integer in base 2
n = int(input("Enter an integer greater than 0: "))
binary = ''
while n > 0:
    rem = n % 2
    binary = str(rem) + binary
    n = n // 2
print(binary)
```

Here we want the quotient and the remainder, not exact division!

Octal to Binary



Hexadecimel to binary

- The hexadecimal or base-16 system (called “hex” for short), which uses 16 different digits, provides a more concise notation than octal for larger numbers.
- Base 16 uses the digits 0 . . . 9 for the corresponding integer quantities and the letters A . . . F for the integer quantities 10 . . . 15

Conversion

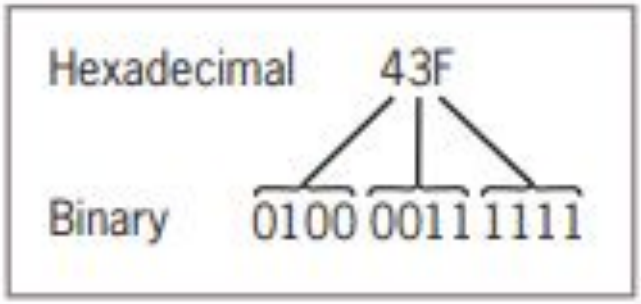


Figure 4-5 The conversion of hexadecimal to binary

Exercises

1. Translate each of the following numbers to decimal numbers:
   a. 11001_2
   b. 100000_2
   c. 11111_2
2. Translate each of the following numbers to binary numbers:
   a. 47_{10}
   b. 127_{10}
   c. 64_{10}

3. Translate each of the following numbers to binary numbers:
   a. 47_8
   b. 127_8
   c. 64_8
4. Translate each of the following numbers to decimal numbers:
   a. 47_8
   b. 127_8
   c. 64_8
5. Translate each of the following numbers to decimal numbers:
   a. 47_{16}
   b. 127_{16}
   c. AA_{16}

String Methods

- Unlike a function, a method is always called with a given data value called an object, which is placed before the method name in the call. The syntax of a method call is the following:

```
<an object>.<method name>(<argument-1>,..., <argument-n>)
```

String Methods in Python

String Method	What it Does
`s.center(width)`	Returns a copy of **s** centered within the given number of columns.
`s.count(sub [, start [, end]])`	Returns the number of non-overlapping occurrences of substring **sub** in **s**. Optional arguments **start** and **end** are interpreted as in slice notation.
`s.endswith(sub)`	Returns **True** if **s** ends with **sub** or **False** otherwise.
`s.find(sub [, start [, end]])`	Returns the lowest index in **s** where substring **sub** is found. Optional arguments **start** and **end** are interpreted as in slice notation.
`s.isalpha()`	Returns **True** if **s** contains only letters or **False** otherwise.
`s.isdigit()`	Returns **True** if **s** contains only digits or **False** otherwise.
`s.join(sequence)`	Returns a string that is the concatenation of the strings in the sequence. The separator between elements is **s**.
`s.lower()`	Returns a copy of **s** converted to lowercase.
`s.replace(old, new [, count])`	Returns a copy of **s** with all occurrences of substring **old** replaced by **new**. If the optional argument **count** is given, only the first **count** occurrences are replaced.
`s.split([sep])`	Returns a list of the words in **s**, using **sep** as the delimiter string. If **sep** is not specified, any whitespace string is a separator.
`s. startswith(sub)`	Returns **True** if **s** starts with **sub** or **False** otherwise.
`s.strip([aString])`	Returns a copy of **s** with leading and trailing whitespace (tabs, spaces, newlines) removed. If **aString** is given, remove characters in **aString** instead.
`s.upper()`	Returns a copy of **s** converted to uppercase.

- help(str.<method name>) to receive documentation on the use of an individual method.
- Note that some arguments in this documentation might be enclosed in square brackets ([]).
- These indicate that the arguments are optional and may be omitted when the method is called

Examples

```
>>> s = "Hi there!"
>>> len(s)
9
>>> s.center(11)
' Hi there! '
>>> s.count('e')
2
>>> s.endswith("there!")
True
>>> s.startswith("Hi")
True
>>> s.find("the")
3
>>> s.isalpha()
False
>>> 'abc'.isalpha()
True
>>> "326".isdigit()
True
>>> words = s.split()
>>> words
['Hi', 'there!']
>>> " ".join(words)
'Hithere!'
>>> " ". join(words)
'Hi there!'
>>> s.lower()
'hi there!'
>>> s.upper()
'HI THERE!'
>>> s.replace('i', 'o')
'Ho there!'
>>> " Hi there! ".strip()
'Hi there!'
```

Exercises

1. Assume that the variable **data** refers to the string **"Python rules!"**. Use a string method from Table 4-2 to perform the following tasks:
   a. Obtain a list of the words in the string.
   b. Convert the string to uppercase.
   c. Locate the position of the string **"rules"**.
   d. Replace the exclamation point with a question mark.

What Is a File?

- A *file* is a software object that allows a program to represent and access data stored in secondary memory

- Two basic types of files:
  - Text files: for storing and accessing text (characters)
  - Binary files: executable programs and their data files (such as images, sound clips, video)

Text Files

- A text file is logically a sequence of characters
- Basic operations are
  - input (read characters from the file)
  - output (write characters to the file)
- There are several flavors of each type of operation

File Input

- We want to bring text in from a file for processing

- Three steps:
  - Open the file for input
  - Read the text and save it in a variable
  - Process the text

Opening a File

```
<a variable> = open(<a file name>, <a flag>)
```

<a flag> can be

'r' - used for input, to *read* from an existing file

'w' - used for output, to *overwrite* an existing file

'a' - used for output, to *append* to an existing file

Example: Read Text from a File

```
filename = input('Enter a file name: ')

myfile = open(filename, 'r')

text = myfile.read()

print(text)
```

The file name must either be in the current directory or be a pathname to a file in a directory

Python raises an error if the file is not found

`text` refers to one big string

Read Lines of Text from a File

```
filename = input('Enter a file name: ')
myfile = open(filename, 'r')

for line in myfile:
    print(line)
```

Extract a substring up to but not including the last character (the newline)

This will produce an exact echo of the input file

Alternatively, Use **`readline`**

```
filename = input('Enter a file name: ')
myfile = open(filename, 'r')

while True:
    line = myfile.readline()
    if line == '':
        break
    print(line)
```

The **`readline`** method reads a line of text and returns it as a string, including the newline character

This method returns the empty string if the end of file is encountered

Count the Words

```
filename = input('Enter a file name: ')
myfile = open(filename, 'r')
wordcount = 0

for line in myfile:
    wordcount += len(line.split())

print('The word count is', wordcount)
```

File Output

- We want to save data to a text file

- Four steps:
  - Open the file for output
  - Covert the data to strings, if necessary
  - Write the strings to the file
  - Close the file

Example: Write Text to a File

```
filename = input('Enter a file name: ')

myfile = open(filename, 'w')

myfile.write('Two lines\nof text')

myfile.close()
```

If the file already exists, it is overwritten; otherwise, it is created in the current directory or path

The **`write`** method expects a string as an argument

Failure to close the file can result in losing data

Example: Write Integers to a File

```
filename = input('Enter a file name: ')

myfile = open(filename, 'w')

for i in range(1, 11):
    myfile.write(str(i) + '\n')

myfile.close()
```

`write` can be called 0 or more times

Data values must be converted to strings before output

Separators, such as newlines or spaces, must be explicitly written as well

Reading numbers from a file

- During input, these data can be read with a simple for loop. This loop accesses a line of text on each pass.
- To convert this line to the integer contained in it, the programmer runs the string method strip to remove the newline and then runs the int function to obtain the integer value

```
f = open("integers.txt", 'r')
theSum = 0
for line in f:
    line = line.strip()
    number = int(line)
    theSum += number
print("The sum is", theSum)
```

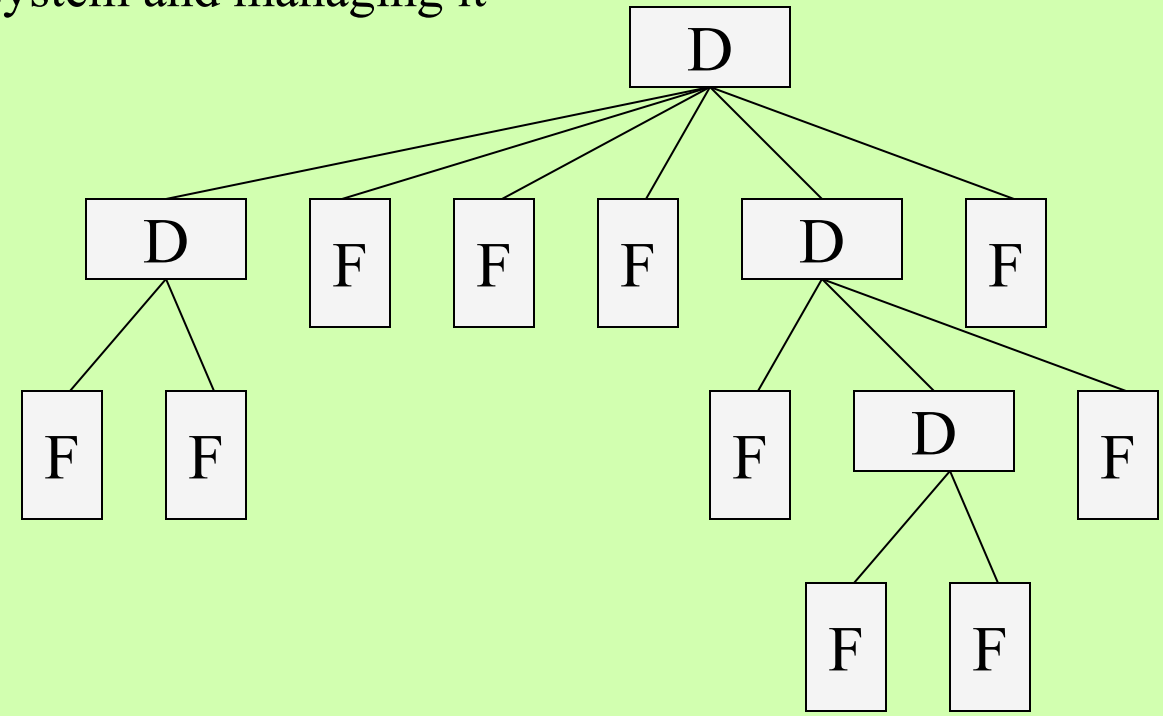
```
f = open("integers.txt", 'r')
theSum = 0
for line in f:
    wordlist = line.split()
    for word in wordlist:
        number = int(word)
        theSum += number
print("The sum is", theSum)
```

File Methods

`open(filename, mode)`	Opens a file at the given filename and returns a file object. The **mode** can be `'r'`, `'w'`, `'rw'`, or `'a'`. The last two values, `'rw'` and `'a'`, mean read/write and append, respectively.
`f.close()`	Closes an output file. Not needed for input files.
`f.write(aString)`	Outputs **aString** to a file.
`f.read()`	Inputs the contents of a file and returns them as a single string. Returns "" if the end of file is reached.
`f.readline()`	Inputs a line of text and returns it as a string, including the newline. Returns "" if the end of file is reached.

Managing Directories

- Directories are organized in a tree-like structure
- Python's **`os`** module includes many functions for navigating through a directory system and managing it



Directories

- The file system of a computer allows you to create folders or directories, within which you can organize files and other directories.
- The complete set of directories and files forms a tree-like structure, with a single root directory at the top and branches down to nested files and subdirectories

- When you launch Python, either from the terminal or from IDLE, the shell is connected to a current working directory.
- At any point during the execution of a program, you can open a file in this directory just by using the file's name.
- You can also access any other file or directory within the computer's file system by using a pathname.
- A file's pathname specifies the chain of directories needed to access a file or directory.
- When the chain starts with the root directory, it's called an **absolute pathname**. When the chain starts from the current working directory, it's called a **relative pathname**

`os` Functions

Function	What It Does
`os.getcwd()`	Returns the current working directory (string)
`os.chdir(path)`	Attaches to the directory specified by path
`os.listdir(path)`	Returns a list of the directory's contents
`os.remove(name)`	Removes a file at path
`os.rename(old, new)`	Resets the old path to the new one
`os.removedirs(path)`	Removes the directory (and all subdirectories) at path

`os.path` Functions

Function	What It Does
`os.path.exists(path)`	Returns **`True`** if **`path`** exists or **`False`** otherwise.
`os.path.isdir(path)`	Returns **`True`** if **`path`** names a directory or **`False`** otherwise.
`os.path.isfile(path)`	Returns **`True`** if **`path`** names a file or **`False`** otherwise.
`os.path.getsize(path)`	Returns the size of the object named by **`path`** in bytes.

Case Study

CASE STUDY: Text Analysis

In 1949, Dr. Rudolf Flesch published *The Art of Readable Writing*, in which he proposed a measure of text readability known as the Flesch Index. This index is based on the average number of syllables per word and the average number of words per sentence in a piece of text. Index scores usually range from 0 to 100, and they indicate readable prose for the following grade levels:

Flesch Index	Grade Level of Readability
0–30	College
50–60	High School
90–100	Fourth Grade

In this case study, we develop a program that computes the Flesch Index for a text file.

Request

Write a program that computes the Flesch Index and grade level for text stored in a text file.

Analysis

The input to this program is the name of a text file. The outputs are the number of sentences, words, and syllables in the file, as well as the file's Flesch Index and Grade Level Equivalent.

During analysis, we consult experts in the problem domain to learn any information that might be relevant in solving the problem. For our problem, this information includes the definitions of *sentence*, *word*, and *syllable*. For the purposes of this program, these terms are defined in Table 4-7.

Word	Any sequence of non-whitespace characters.
Sentence	Any sequence of words ending in a period, question mark, exclamation point, colon, or semicolon.
Syllable	Any word of three characters or less; or any vowel (a, e, i, o, u) or pair of consecutive vowels, except for a final *-es*, *-ed*, or *-e* that is not *-le*.

Cntd:

Note that the definitions of *word* and *sentence* are approximations. Some words, such as *doubles* and *kettles*, end in *-es* but will be counted as having one syllable, and an ellipsis (...) will be counted as three syllables.

Flesch's formula to calculate the index *F* is the following:

$$F = 206.835 - 1.015 \times (words / sentences) - 84.6 \times (syllables / words)$$

The Flesch-Kincaid Grade Level Formula is used to compute the Equivalent Grade Level *G*:

$$G = 0.39 \times (words / sentences) + 11.8 \times (syllables / words) - 15.59$$

Design

This program will perform the following tasks:

1. Receive the filename from the user, open the file for input, and input the text.
2. Count the sentences in the text.
3. Count the words in the text.
4. Count the syllables in the text.
5. Compute the Flesch Index.
6. Compute the Grade Level Equivalent.
7. Print these two values with the appropriate labels, as well as the counts from tasks 2–4.

Task	What it Does
count the sentences	Counts the number of sentences in **text**.
count the words	Counts the number of words in **text**.
count the syllables	Counts the number of syllables in **text**.
compute the Flesch Index	Computes the Flesch Index for the given numbers of sentences, words, and syllables.
compute the grade level	Computes the Grade Level Equivalent for the given numbers of sentences, words, and syllables.

Table 4-8 The tasks defined in the text analysis program

The last task is the most complex. For each word in the text, we must count the syllables in that word. From analysis, we know that each distinct vowel counts as a syllable, unless it is in the endings *-ed*, *-es*, or *-e* (but not *-le*). For now, we ignore the possibility of consecutive vowels.

Cntd:

Implementation (Coding)

The main tasks are marked off in the program code with a blank line and a comment.

```
"""
Program: textanalysis.py
Author: Ken
Computes and displays the Flesch Index and the Grade
Level Equivalent for the readability of a text file.
"""

# Take the inputs
fileName = input("Enter the file name: ")
inputFile = open(fileName, 'r')
text = inputFile.read()

# Count the sentences
sentences = text.count('.') + text.count('?') + \
            text.count(':') + text.count(';') + \
            text.count('!')

# Count the words
words = len(text.split())
```

Cntd:

```
# Count the syllables
syllables = 0
vowels = "aeiouAEIOU"
for word in text.split():
    for vowel in vowels:
        syllables += word.count(vowel)
    for ending in ['es', 'ed', 'e']:
        if word.endswith(ending):
            syllables -= 1
    if word.endswith('le'):
        syllables += 1

# Compute the Flesch Index and Grade Level
index = 206.835 - 1.015 * (words / sentences) - \
        84.6 * (syllables / words)
level = round(0.39 * (words / sentences) + 11.8 * \
        (syllables / words) - 15.59)

# Output the results
print("The Flesch Index is", index)
print("The Grade Level Equivalent is", level)
print(sentences, "sentences")
print(words, "words")
print(syllables, "syllables")
```

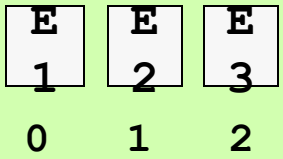
Testing

Although the main tasks all collaborate in the text analysis program, they can be tested more or less independently, before the entire program is tested. After all, there is no point in running the complete program if you are unsure that even one of the tasks does not work correctly.

This kind of procedure is called **bottom-up testing**. Each task is coded and tested before it is integrated into the overall program. After you have written code for one or two tasks, you can test them in a short script. This script is called a **driver**. For

What Is a List?

- A *list* is a sequence of 0 or more data values (of any types) called *elements*



- The programmer can access or replace the element at any position in a list

- An element can be inserted at any position or removed from any position

Real-World Examples

- A shopping list
- A schedule of athletic contests
- A team roster
- An algorithm (a list of instructions)

Literals, Assignment, Comparisons, Concatenation, `for` Loop

```
a = [1, 2, 3]

b = list(range(1, 4))   # b now refers to [1, 2, 3]

a == b                  # returns True
a < [2, 3, 4]           # returns True

print(a + b)            # displays [1, 2, 3, 1, 2, 3]

print(len(a))           # displays 3

for item in [1, 2, 3]: print(item)
```

Similar to the behavior of strings so far

Indexing and Slicing

```
a = [1, 2, 3]

print(a[2])                 # Displays 3

print(a[len(a) - 1])        # Displays 3

print(a[-1])                # Displays 3

print(a[0:2])               # Displays [1, 2]
```

Similar to the behavior of strings so far

Operators and functions in List

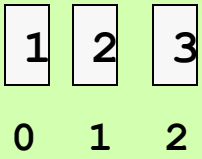
Operator or Function	What It Does
`L[<an integer expression>]`	Subscript used to access an element at the given index position.
`L[<start>:<end>]`	Slices for a sublist. Returns a new list.
`L1 + L2`	List concatenation. Returns a new list consisting of the elements of the two operands.
`print(L)`	Prints the literal representation of the list.
`len(L)`	Returns the number of elements in the list.
`list(range(<upper>))`	Returns a list containing the integers in the range **0** through **`upper - 1`**.
`==, !=, <, >, <=, >=`	Compares the elements at the corresponding positions in the operand lists. Returns **`True`** if all the results are true, or **`False`** otherwise.
`for <variable> in L: <statement>`	Iterates through the list, binding the variable to each element.
`<any value> in L`	Returns **`True`** if the value is in the list or **`False`** otherwise.

Three Ways to Get a Sum

```
a = [1, 2, 3]

total = 0
for index in range(len(a)):
    total += a[index]


total = 0
for item in a:
    total += item


total = sum(a)
```

Replacing an Item

```
[1, 2, 3]
```



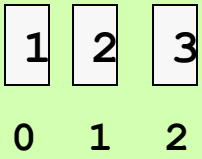
To replace an item at a given position, use the subscript operator with the appropriate index

```
<a list>[<an int>] = <an expression>

a = [1, 2, 3]

a[1] = 5          # The list is now [1, 5, 3]

print(a[1])       # Displays 5
```

Unlike strings and tuples, lists are mutable!

Increment ‘Em All

`[1, 2, 3]`



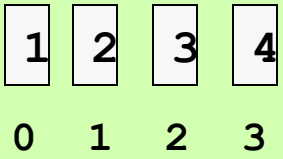

```
a = [1, 2, 3]

for index in range(len(a)):
    a[index] = a[index] + 1
```

Cannot use an item-based **`for`** loop for replacements

Must use an index-based loop

Replacing a Subsequence

```
[1, 2, 3, 4]
```




```
a = [1, 2, 3, 4]

a[0:2] = [5, 6]

print(a)                    # Displays [5, 6, 3, 4]
```

Splitting

`split` builds a list of tokens (words) from a string using the space or newline as the default separator

```
s = 'Python is way cool!'

lyst = s.split()

print(lyst)                    # Displays ['Python', 'is', 'way', 'cool!']
```

<a string>.split(*<optional separator string>*)

Example

- The program uses the string method split to extract a list of the words in a sentence. These words are then converted to uppercase letters within the list

```
>>> sentence = "This example has five words."
>>> words = sentence.split()
>>> words
['This', 'example', 'has', 'five', 'words.']
>>> for index in range(len(words)):
        words[index] = words[index].upper()
>>> words
['THIS', 'EXAMPLE', 'HAS', 'FIVE', 'WORDS.']
```

List Methods

List Method	What It Does
`L.append(element)`	Adds **element** to the end of **L**.
`L.extend(aList)`	Adds the elements of **aList** to the end of **L**.
`L.insert(index, element)`	Inserts **element** at **index** if **index** is less than the length of **L**. Otherwise, inserts **element** at the end of **L**.
`L.pop()`	Removes and returns the element at the end of **L**.
`L.pop(index)`	Removes and returns the element at **index**.

Insert

- When the index is less than the length of the list, this method places the new element before the existing element at that index, after shifting elements to the right by one position
- When the index is greater than or equal to the length of the list, the new element is added to the end of the list

```
>>> example = [1, 2]
>>> example
[1, 2]
>>> example.insert(1, 10)
>>> example
[1, 10, 2]
>>> example.insert(3, 25)
>>> example
[1, 10, 2, 25]
```

append

- append expects just the new element as an argument and adds the new element to the end of the list.

```
>>> example = [1, 2]
>>> example
[1, 2]
>>> example.append(3)
>>> example
[1, 2, 3]
>>> example.extend([11, 12, 13])
>>> example
[1, 2, 3, 11, 12, 13]
>>> example + [14, 15]
[1, 2, 3, 11, 12, 13, 14, 15]
>>> example
[1, 2, 3, 11, 12, 13]
```

pop

```
>>> example
[1, 2, 10, 11, 12, 13]
>>> example.pop()          # Remove the last element
13
>>> example
[1, 2, 10, 11, 12]
>>> example.pop(0)         # Remove the first element
1
>>> example
[2, 10, 11, 12]
```

Searching in list

- The **in** operator determines an element's presence or absence, use the method index to locate an element's position in a list.
- It is unfortunate that index raises an exception when the target element is not found.
- To guard against this unpleasant consequence, you must first use the in operator to test for presence and then the index method if this test returns True

Example

```
aList = [34, 45, 67]
target = 45
if target in aList:
    print(aList.index(target))
else:
    print(-1)
```

Sorting

```
>>> example = [4, 2, 10, 8]
>>> example
[4, 2, 10, 8]
>>> example.sort()
>>> example
[2, 4, 8, 10]
```

Mutator Methods and value None

- Mutable objects (such as lists) have some methods devoted entirely to modifying the internal state of the object. Such methods are called mutators.
- Examples are the list methods insert, append, extend, pop, and sort.
- Because a change of state is all that is desired, a mutator method usually returns no value of interest to the caller (but note that pop is an exception to this rule).
- Python nevertheless automatically returns the special value **None** even when a method does not explicitly return a value

Example

```
>>> alist=alist.sort()
>>>print(alist)
None
```

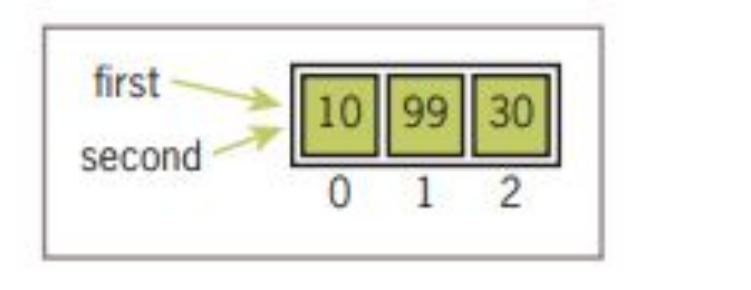
Aliasing and Side effects

```
>>> first = [10, 20, 30]
>>> second = first
>>> first
[10, 20, 30]
>>> second
[10, 20, 30]
>>> first[1] = 99
>>> first
```

```
[10, 99, 30]
>>> second
[10, 99, 30]
```

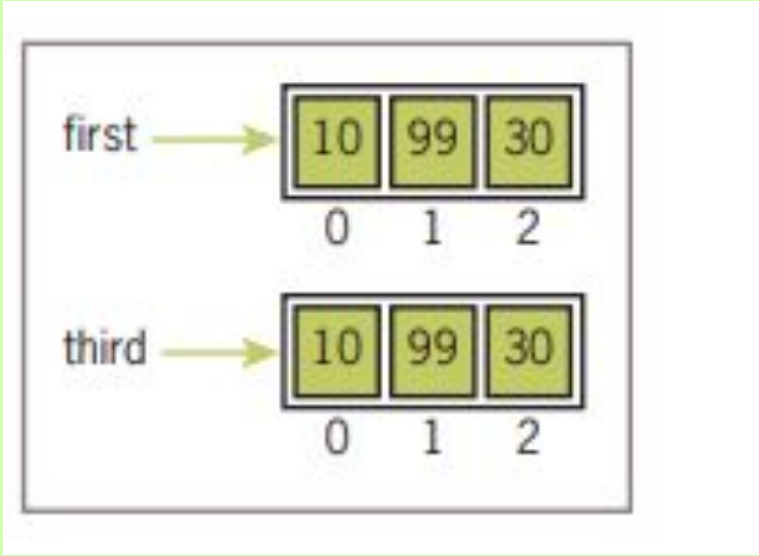
- When the second element of the list named first is replaced, the second element of the list named second is replaced also.
- This type of change is what is known as a **side effect**.
- This happens because after the assignment second = first, the variables first and second refer to the exact same list object.
- They are **aliases** for the same object

- Aliasing



- To prevent aliasing, you can create a new object and copy the contents of the original to it

```
>>> third = []
>>> for element in first:
        third.append(element)
>>> first
[10, 99, 30]
>>> third
[10, 99, 30]
>>> first[1] = 100
>>> first
[10, 100, 30]
>>> third
[10, 99, 30]
```

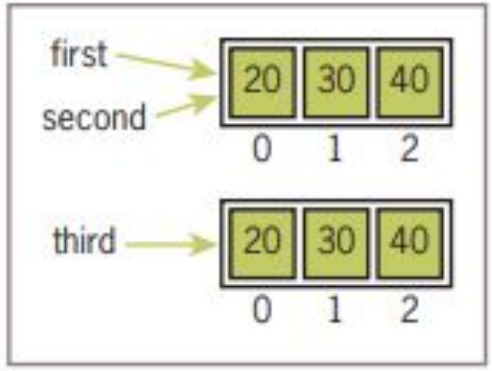


Object Identity

- The == operator returns True if the variables are aliases for the same object.
- Unfortunately, == also returns True if the contents of two different objects are the same.
- The first relation is called **object identity**, whereas the second relation is called **structural equivalence**.
- The == operator has no way of distinguishing between these two types of relations.
- Python's **is operator** can be used to test for object identity. It returns True if the two operands refer to the exact same object, and it returns False if the operands refer to distinct objects (even if they are structurally equivalent)

Example

```
>>> first = [20, 30, 40]
>>> second = first
>>> third = list(first)          # Or first[:]
>>> first == second
True
>>> first == third
True
>>> first is second
True
>>> first is third
False
```



Median f numbers from File

```
fileName = input("Enter the filename: ")
f = open(fileName, 'r')

# Input the text, convert it to numbers, and
# add the numbers to a list
numbers = []
for line in f:
    words = line.split()
    for word in words:
        numbers.append(float(word))

# Sort the list and print the number at its midpoint
numbers.sort()
midpoint = len(numbers) // 2
print("The median is", end = " ")
if len(numbers) % 2 == 1:
    print(numbers[midpoint])
else:
    print((numbers[midpoint] + numbers[midpoint - 1]) / 2)
```

Tuple

- A tuple is a type of sequence that resembles a list, except that, unlike a list, a tuple is immutable.
- You indicate a tuple literal in Python by enclosing its elements in parentheses instead of square brackets

```
>>> fruits = ("apple", "banana")
>>> fruits
('apple', 'banana')
```

```
>>> meats = ("fish", "poultry")
>>> meats
('fish', 'poultry')
>>> food = meats + fruits
>>> food
('fish', 'poultry', 'apple', 'banana')
>>> veggies = ["celery", "beans"]
>>> tuple(veggies)
('celery', 'beans')
```

Case Study

CASE STUDY: Generating Sentences

Can computers write poetry? We'll attempt to answer that question in this case study by giving a program a few words to play with.

Request

Write a program that generates sentences.

Analysis

Sentences in any language have a structure defined by a **grammar**. They also include a set of words from the **vocabulary** of the language. The vocabulary of a language like English consists of many thousands of words, and the grammar rules are quite complex. For the sake of simplicity our program will generate sentences from a simplified subset of English. The vocabulary will consist of sample words from several parts of speech, including nouns, verbs, articles, and prepositions. From these words, you can build noun phrases, prepositional phrases, and verb phrases. From these constituent phrases, you can build sentences. For example, the sentence "The girl hit the ball with the bat" contains three noun phrases, one verb phrase, and one prepositional phrase. Table 5-3 summarizes the grammar rules for our subset of English.

Phrase	Its Constituents
Sentence	Noun phrase + Verb phrase
Noun phrase	Article + Noun
Verb phrase	Verb + Noun phrase + Prepositional phrase
Prepositional phrase	Preposition + Noun phrase

Table 5-3 The grammar rules for the sentence generator

Cntd:

Design

Of the many ways to solve the problem in this case study, perhaps the simplest is to assign the task of generating each phrase to a separate function. Each function builds and returns a string that represents its phrase. This string contains words drawn from the parts of speech and from other phrases. When a function needs an individual word, it is selected at random from the words in that part of speech. When a function needs another phrase, it calls another function to build that phrase. The results, all strings, are concatenated with spaces and returned.

The function for **Sentence** is the easiest. It just calls the functions for **Noun phrase** and **Verb phrase** and concatenates the results, as in the following:

```
def sentence():
    """Builds and returns a sentence."""
    return nounPhrase() + " " + verbPhrase() + "."
```

The function for **Noun phrase** picks an article and a noun at random from the vocabulary, concatenates them, and returns the result. We assume that the variables **articles** and **nouns** refer to collections of these parts of speech and develop these later in the design. The function **random.choice** returns a random element from such a collection.

```
def nounPhrase() :
    """Builds and returns a noun phrase."""
    return random.choice(articles) + " " + random.choice(nouns)
```

The design of the remaining two phrase-structure functions is similar.

The **main** function drives the program with a count-controlled loop:

```
def main():
   """Allows the user to input the number of sentences to generate."""
   number = int(input("Enter the number of sentences: "))
   for count in range(number):
       print(sentence())
```

The variables **articles** and **nouns** used in the program's functions refer to the collections of actual words belonging to these two parts of speech. Two other collections, named **verbs** and **prepositions**, also will be used. The data structure used to represent a collection of words should allow the program to pick one word at random.

Cntd:

```
import random

# Vocabulary: words in 4 different parts of speech
articles     = ("A", "THE")
nouns        = ("BOY", "GIRL", "BAT", "BALL")
verbs        = ("HIT", "SAW", "LIKED")
prepositions = ("WITH", "BY")

def sentence():
    """Builds and returns a sentence."""
    return nounPhrase() + " " + verbPhrase()

def nounPhrase():
    """Builds and returns a noun phrase."""
    return random.choice(articles) + " " + random.choice(nouns)

def verbPhrase():
    """Builds and returns a verb phrase."""
    return random.choice(verbs) + " " + nounPhrase() + " " + \
           prepositionalPhrase()

def prepositionalPhrase():
    """Builds and returns a prepositional phrase."""
    return random.choice(prepositions) + " " + nounPhrase()

def main():
    """Allows the user to input the number of sentences
    to generate."""
    number = int(input("Enter the number of sentences: "))
```

Cntd:

```
    for count in range(number):
        print(sentence())

# The entry point for program execution
if __name__ == "__main__":
    main()
```

Testing

Poetry it's not, but testing is still important. The functions developed in this case study can be tested in a bottom-up manner. To do so, you must initialize the data first. Then you can run the lowest-level function, **nounPhrase**, immediately to check its results, and you can work up to sentences from there.

On the other hand, testing can also follow the design, which took a top-down path. You might start by writing headers for all of the functions and simple **return** statements that return the functions' names. Then you can complete the code for the **sentence** function first, test it, and proceed downward from there. The wise programmer can also mix bottom-up and top-down testing as needed.

Data Structures

- A *data structure* is a means of organizing several data elements so they can be treated as one thing
- A *sequence* is a data structure in which the elements are accessible by *position* (first .. last)
- A *dictionary* is a data structure in which the elements are accessible by *content*

Examples of Dictionaries

- Dictionary
- Phone book
- Thesaurus
- Encyclopedia
- Cookbook
- World Wide Web

Elements may also be ordered alphabetically, but they need not be

Examples of Dictionaries

- Dictionary
- Phone book
- Thesaurus
- Encyclopedia
- Cookbook
- World Wide Web

An element is accessed by *content*

This content can be

- A word
- A person's name
- A food type
- A text phrase or an image

Examples of Dictionaries

- Dictionary
- Phone book
- Thesaurus
- Encyclopedia
- Cookbook
- World Wide Web

An element is accessed by *content*

This content can be

- A word
- A person's name
- A food type
- A text phrase or an image

Each content is called a *key*

Each associated element is called a *value*

Characteristics of a Dictionary

- A dictionary is a *set* of keys associated with values
- The keys are unique and need not be ordered by position or alphabetically
- Values can be duplicated
- Keys and values can be of any data types

Examples of Keys and Values

Some hexadecimal ($base_{16}$) digits and their values

`'A'`	`10`
`'B'`	`11`
`'C'`	`12`
`'D'`	`13`
`'E'`	`14`
`'F'`	`15`

A database of Ken's info

`'name'`	`'Ken'`
`'age'`	`67`
`'gender'`	`'M'`
`'occupation'`	`'teacher'`
`'hobbies'`	`['movies', 'gardening']`

Dictionaries in Python

```
hexdigits = {'A':10, 'B':11, 'C':12,
             'D':13, 'E':14, 'F':15}

database = {'name':'Ken', 'age':67,
            'gender':'M', 'occupation':'teacher'}

an_empty_one = {}
```

Syntax:

```
{<key> : <value>, … , <key> : <value>}
```

Adding an Replacing key values

You add a new key/value pair to a dictionary by using the subscript operator `[]`. The form of this operation is the following:

```
<a dictionary>[<a key>] = <a value>
```

The next code segment creates an empty dictionary and adds two new entries:

```
>>> info = {}
>>> info["name"] = "Sandy"
>>> info["occupation"] = "hacker"
>>> info
{'name':'Sandy', 'occupation':'hacker'}
```

The subscript is also used to replace a value at an existing key, as follows:

```
>>> info["occupation"] = "manager"
>>> info
{'name':'Sandy', 'occupation':'manager'}
```

Acessing a Value with a Key

```
>>> database = {'name':'Ken', 'age':67,
                'gender':'M', 'occupation':'teacher'}

>>> database['name']
'Ken'

>>> database['age']
67
```

The subscript expects a key in the dictionary and returns the associated value

```
<dictionary>[<key>]
```

Key Must be in the Dictionary

```
>>> database = {'name':'Ken', 'age':67,
                'gender':'M', 'occupation':'teacher'}

>>> database['hair color']

Traceback (most recent call last):
  File "<pyshell#6>", line 1, in -toplevel-
    database['hair color']
KeyError: 'hair color'
```

Guard Access with an `if`

```
>>> database = {'name':'Ken', 'age':67,
                'gender':'M', 'occupation':'teacher'}

>>> if 'hair color' in database:
        database['hair color']
```

The **`in`** operator can be used to search any sequence or dictionary

Alternative: Use the **`get`** Method

```
>>> database = {'name':'Ken', 'age':67,
                'gender':'M', 'occupation':'teacher'}

>>> database.get('hair color', None)
None
```

If the key (first argument) is in the dictionary, the value is returned

Otherwise, the default value (second argument) is returned

The `for` Loop Visits All Keys

```
>>> database = {'name':'Ken', 'age':67,
                'gender':'M', 'occupation':'teacher'}

>>> for key in database:
        print(key, database[key])
gender M
age 67
name Ken
occupation teacher
```

Inserting a New Key/Value

```
>>> database = {'name':'Ken', 'age':67,
                'gender':'M', 'occupation':'teacher'}

>>> database['hair color'] = 'gray'
```

If the key is not already in the dictionary, Python creates one and inserts it with the associated value

Inserting a New Key/Value

```
hexdigits = {'A':10, 'B':11, 'C':12,
             'D':13, 'E':14, 'F':15}


>>> for i in range(10):
        hexdigits[str(i)] = i

>>> hexdigits
{'A':10, 'B':11, 'C':12, 'D':13, 'E':14, 'F':15,
 '1': 1, '0': 0, '3': 3, '2': 2, '5': 5, '4': 4,
 '7': 7, '6': 6, '9': 9, '8': 8}
```

Insert the remaining hexadecimal digits and their integer values

Replacing an Existing Value

```
>>> database = {'name':'Ken', 'age':67,
                'gender':'M', 'occupation':'teacher'}

>>> database['age'] = database['age'] + 1
```

If the key is already in the dictionary, Python replaces the associated value with the new one

Removing a key

```
>>> database = {'name':'Ken', 'age':67,
                'gender':'M', 'occupation':'teacher'}

>>> database.pop('age', None)
67
```

The **pop** method removes the key and its associated value and returns this value

Dictionary Operation	What It Does
`len(d)`	Returns the number of entries in **d**.
`d[key]`	Used for inserting a new key, replacing a value, or obtaining a value at an existing key.
`d.get(key [, default])`	Returns the value if the key exists or returns the default if the key does not exist. Raises an error if the default is omitted and the key does not exist.
`d.pop(key [, default])`	Removes the key and returns the value if the key exists or returns the default if the key does not exist. Raises an error if the default is omitted and the key does not exist.
`list(d.keys())`	Returns a list of the keys.
`list(d.values())`	Returns a list of the values.
`list(d.items())`	Returns a list of tuples containing the keys and values for each entry.
`d.clear()`	Removes all the keys.
`for key in d:`	**key** is bound to each key in **d** in an unspecified order.

Case study

CASE STUDY: Nondirective Psychotherapy

In the early 1960s, the MIT computer scientist Joseph Weizenbaum developed a famous program called ELIZA that could converse with the computer user, mimicking a nondirective style of psychotherapy. The doctor in this kind of therapy is essentially a good listener who responds to the patient's statements by rephrasing them or indirectly asking for more information. To illustrate the use of data structures, we develop a drastically simplified version of this program.

Request

Write a program that emulates a nondirective psychotherapist.

Cntd:

Request

Write a program that emulates a nondirective psychotherapist.

Analysis

Figure 5-4 shows the program's interface as it changes throughout a sequence of exchanges with the user.

```
Good morning, I hope you are well today.
What can I do for you?

>> My mother and I don't get along
Why do you say that your mother and you don't get along

>> she always favors my sister
You seem to think that she always favors your sister

>> my dad and I get along fine
Can you explain why your dad and you get along fine

>> he helps me with my homework
Please tell me more

>> quit
Have a nice day!
```

Figure 5-4 A session with the doctor program

When the user enters a statement, the program responds in one of two ways:

1. With a randomly chosen hedge, such as "Please tell me more."
2. By changing some key words in the user's input string and appending this string to a randomly chosen qualifier. Thus, to "My teacher always plays favorites," the program might reply, "Why do you say that your teacher always plays favorites?"

Cntd:

Design

The program consists of a set of collaborating functions that share a common data pool.

Two of the data sets are the hedges and the qualifiers. Because these collections do not change and their elements must be selected at random, you can use tuples to represent them. Their names, of course, are **`hedges`** and **`qualifiers`**.

The other set of data consists of mappings between first-person pronouns and second-person pronouns. For example, when the program sees "I" in a patient's input, it should respond with a sentence containing "you." The best type of data structure to hold these correlations is a dictionary. This dictionary is named **`replacements`**.

The **`main`** function displays a greeting, displays a prompt, and waits for user input. The following is pseudocode for the main loop:

```
output a greeting to the patient
while True
    prompt for and input a string from the patient
    if the string equals "Quit"
        output a sign-off message to the patient
        break
    call another function to obtain a reply to this string
    output the reply to the patient
```

Our therapist might not be an expert, but there is no charge for its services. What's more, our therapist seems willing to go on forever. However, if the patient must quit to do something else, she can do so by typing "quit" to end the program.

The **reply** function expects the patient's string as an argument and returns another string as the reply. This function implements the two strategies for making replies suggested in the analysis phase. A quarter of the time a hedge is warranted. Otherwise, the function constructs its reply by changing the persons in the patient's input and appending the result to a randomly selected qualifier. The **reply** function calls yet another function, **changePerson**, to perform the complex task of changing persons.

The **changePerson** function extracts a list of words from the patient's string. It then builds a new list wherein any pronoun key in the replacements dictionary is replaced by its pronoun/value. This list is then converted back to a string and returned.

Implementation (Coding)

The structure of this program resembles that of the sentence generator developed in the first case study of this chapter. The three data structures are initialized near the beginning of the program, and they never change. The three functions collaborate in a straightforward manner. Here is the code:

```
"""
Program: doctor.py
Author: Ken
Conducts an interactive session of nondirective
psychotherapy.
"""

import random

hedges = ("Please tell me more.",
          "Many of my patients tell me the same thing.",
          "Please continue.")

qualifiers = ("Why do you say that ",
              "You seem to think that ",
              "Can you explain why ")

replacements = {"I":"you", "me":"you", "my":"your",
                "we":"you", "us":"you", "mine":"yours"}
```

```
def reply(sentence):
    """Builds and returns a reply to the sentence."""
    probability = random.randint(1, 4)
    if probability == 1:
        return random.choice(hedges)
    else:
        return random.choice(qualifiers) + changePerson(sentence)

def changePerson(sentence):
    """Replaces first person pronouns with second person
    pronouns."""
    words = sentence.split()
    replyWords = []
    for word in words:
        replyWords.append(replacements.get(word, word))
    return " ".join(replyWords)

def main():
    """Handles the interaction between patient and doctor."""
    print("Good morning, I hope you are well today.")
    print("What can I do for you?")
    while True:
        sentence = input("\n>> ")
        if sentence.upper() == "QUIT":
            print("Have a nice day!")
            break
        print(reply(sentence))

# The entry point for program execution
if __name__ == "__main:"__
    main()
```

Cntd:

Testing

As in the sentence-generator program, the functions in this program can be tested in a bottom-up or a top-down manner. As you will see, the program's replies break down when the user addresses the therapist in the second person, when the user inputs contractions (for example, I'm and I'll), when the user addresses the doctor directly with sentences like "You are not listening to me," and in many other ways. As you'll see in the Projects at the end of this chapter, with a little work you can make the replies more realistic.

Exercises

For questions 1–6, assume that the variable **data** refers to the list **[10, 20, 30]**.

1. The expression **data[1]** evaluates to

   a. **10** b. **20**

2. The expression **data[1:3]** evaluates to

   a. **[10, 20, 30]** b. **[20, 30]**

3. The expression **data.index(20)** evaluates to

   a. **1**
   b. **2**
   c. **True**

4. The expression **data + [40, 50]** evaluates to

   a. **[10, 60, 80]** b. **[10, 20, 30, 40, 50]**

5. After the statement **data[1] = 5, data** evaluates to

   a. **[5, 20, 30]** b. **[10, 5, 30]**

6. After the statement **data.insert(1, 15)**, the original **data** evaluates to

   a. **[15, 10, 20, 30]**
   b. **[10, 15, 30]**
   c. **[10, 15, 20, 30]**

For questions 7–9, assume that the variable **info** refers to the dictionary **{"name":"Sandy", "age":17}**.

7. The expression **list(info.keys())** evaluates to

   a. **("name", "age")** b. **["name", "age"]**

8. The expression **info.get("hobbies", None)** evaluates to

   a. **"knitting"**
   b. **None**
   c. **1000**

Cntd:

9. The method to remove an entry from a dictionary is named
   a. `delete`
   b. `pop`
   c. `remove`
10. Which of the following are immutable data structures?
    a. dictionaries and lists
    b. strings and tuples

Why Design?

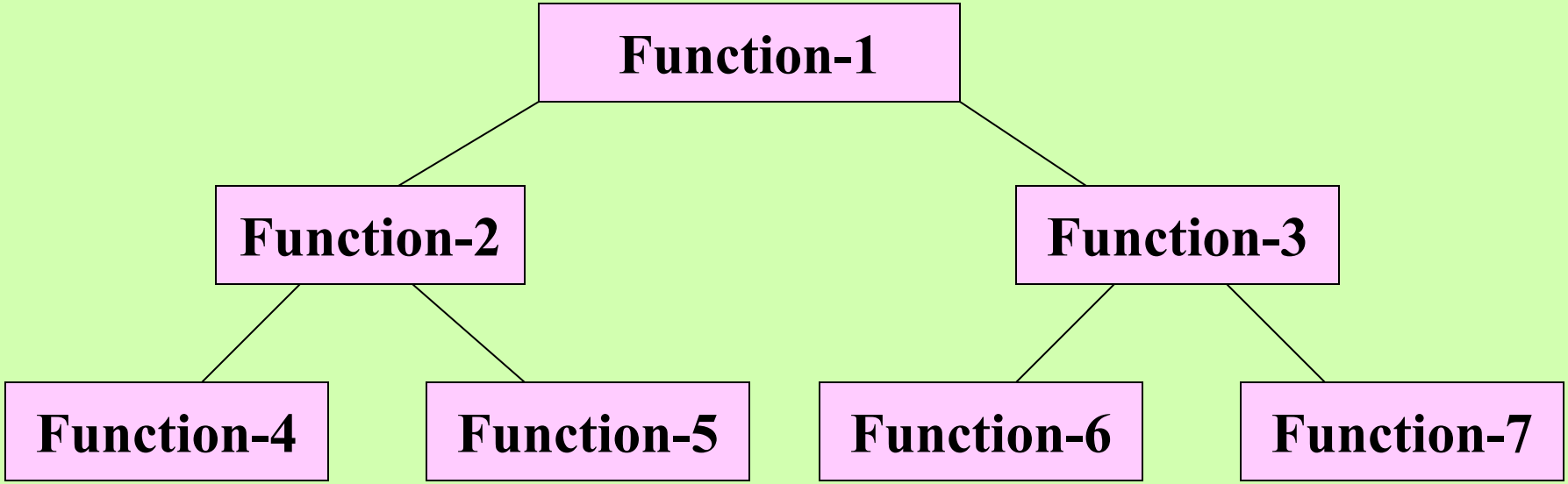
- As problems become more complex, so do their solutions
- There is more to programming than just hacking out code
- We can decompose a complex problem into simpler subproblems and solve each of these
- Divide up the work and assign responsibilities to individual actors (functions)

Functions as Abstraction Mechanism

- Functions eliminate redundancy
- Functions hide complexity
- Functions support general methods with systematic variations
- Functions support the division of labour

Top-Down Design

In top-down design, we decompose a complex problem into simpler subproblems, and solve these with different functions.



Top down design

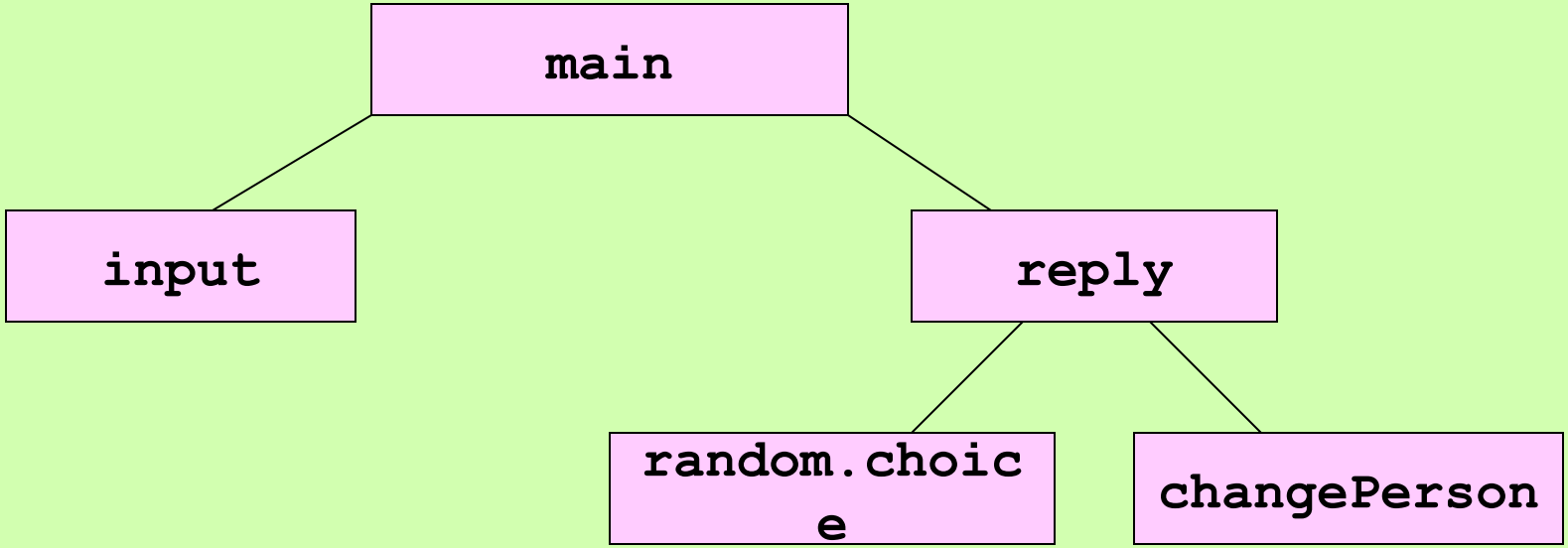
- This strategy starts with a global view of the entire problem and breaks the problem into smaller, more manageable subproblems—a process known as **problem decomposition**.
- As each subproblem is isolated, its solution is assigned to a function.
- Problem decomposition may continue down to lower levels, because a subproblem might in turn contain two or more lower-level problems to solve.

Top down design

- As functions are developed to solve each subproblem, the solution to the overall problem is gradually filled out in detail. This process is also called **stepwise refinement**

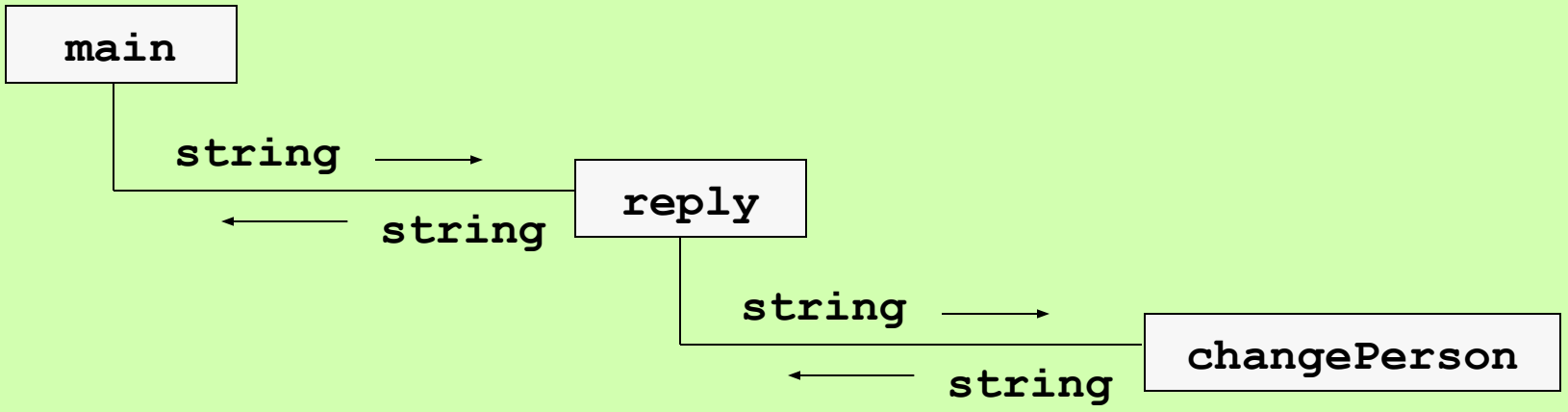
Example: The **`doctor`** Program

Each function has its own responsibilities; in a well-designed program, no function does too much



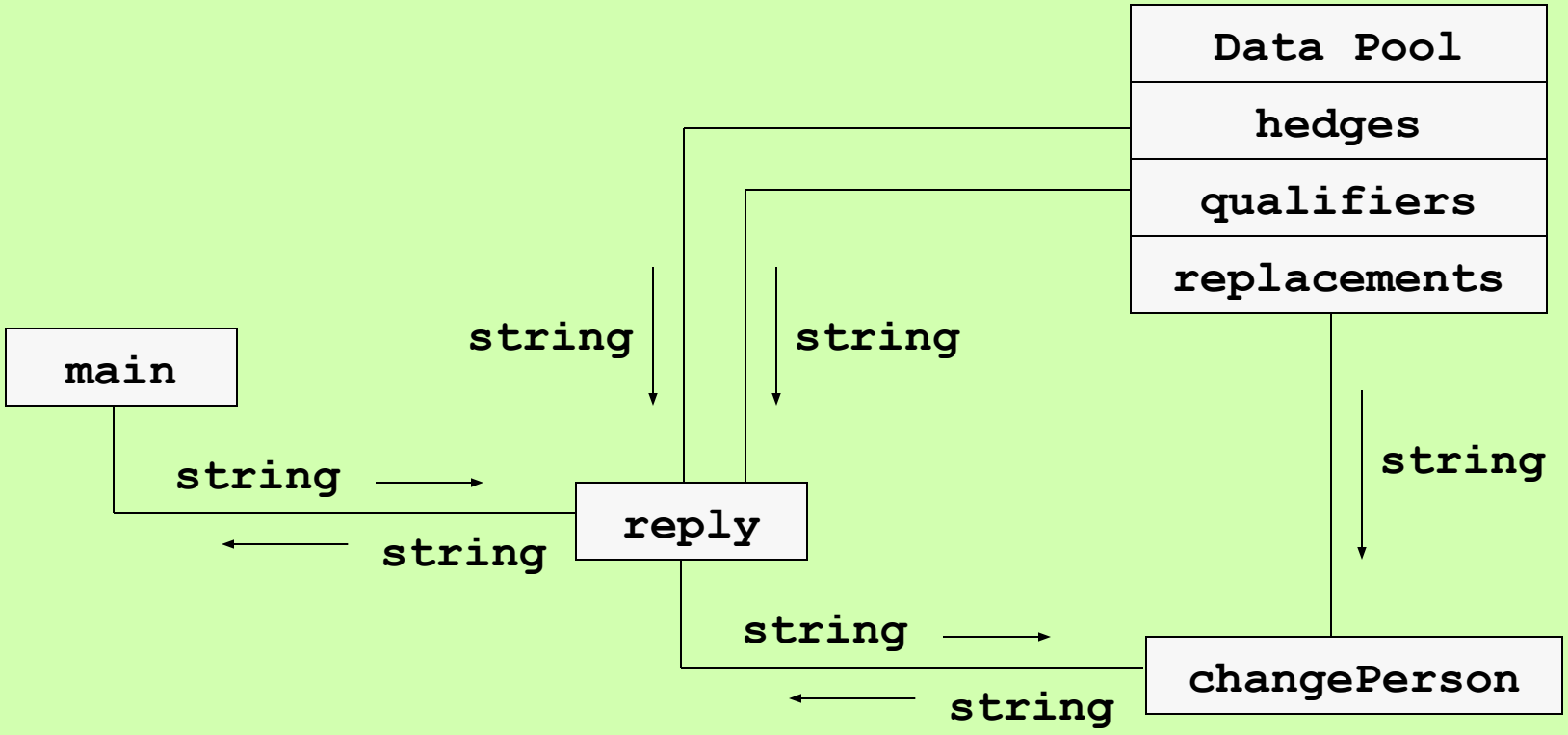
Example: The **`doctor`** Program

Functions communicate via arguments and returned values



Example: The **doctor** Program

Functions also go to shared data pools for information



Example: The Sentence Program

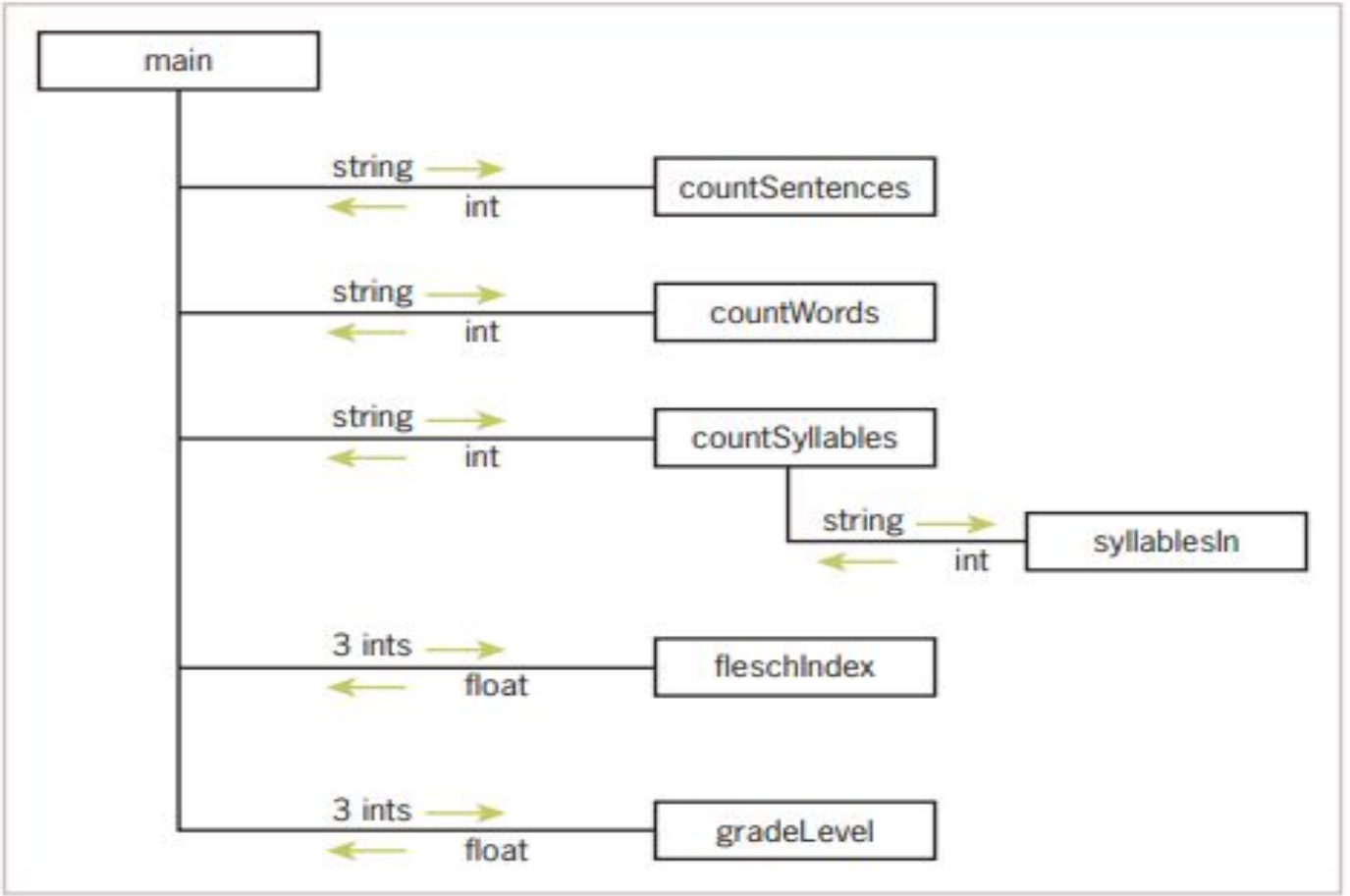
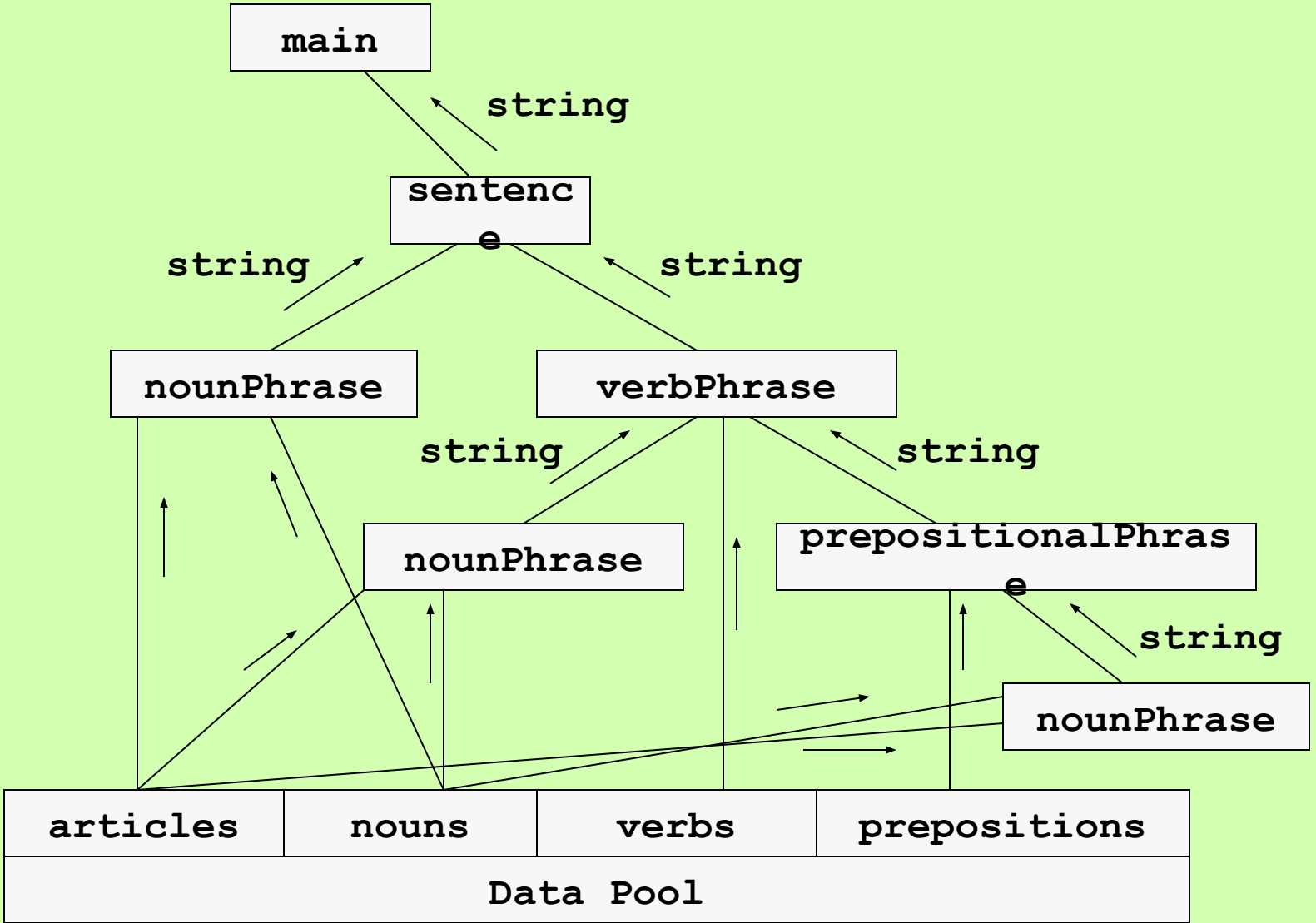


Figure 6-1 A structure chart for the text-analysis program

Example: The Sentence Program



Design Strategies: Top Down

- Start with the **`main`** function and pretend that the functions that it calls are already defined
- Work your way down by defining those functions, etc.
- Cannot test anything until they're all finished

Drivers and Stubs

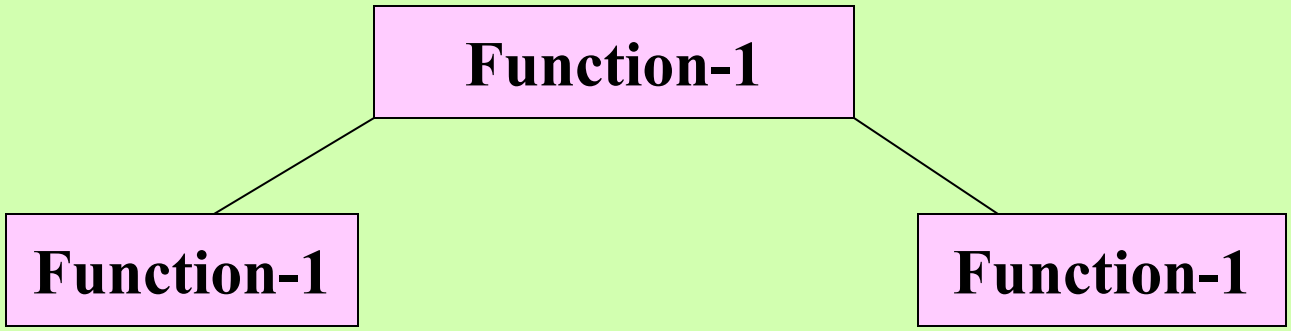
- Start with the **`main`** function and pretend that the functions that it calls are already defined
- Define these functions using simple headers and almost no code
  - If a function returns a value, return a reasonable default (0 or empty string)
  - If a function does not return a value, return **`None`**
- The **`main`** function is known as a *driver*, and the other functions are called *stubs*

Design Strategies: Bottom Up

- Start with simple functions at the bottom of the chart and work your way up
- Each new function can be tested as soon as it's defined
- Easier to get the little pieces of a program up and running, then integrate them into a more complete solution

Recursive Design

As a special case of top-down design, we decompose a problem into smaller subproblems that have the *same form*, and solve these with the *same function*.



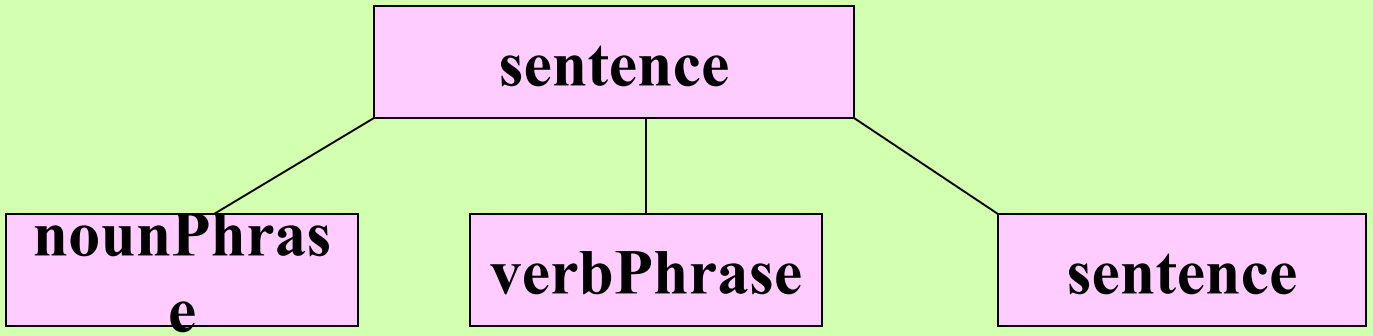
Recursive function

- A recursive function is a function that calls itself.
- To prevent a function from repeating itself indefinitely, it must contain at least one selection statement.
- This statement examines a condition called a **base case** to determine whether to stop or to continue with another recursive step

Recursive Design

As a special case, we decompose a problem into smaller subproblems that have the *same form*, and solve these with the *same function*.

sentence = nounPhrase verbPhrase [“and” sentence]



Example: Displayrange

- Let's examine how to convert an iterative algorithm to a recursive function. Here is a definition of a function displayRange that prints the numbers from a lower bound to an upper bound:

```
def displayRange(lower, upper):
    """Outputs the numbers from lower through upper."""
    while lower <= upper:
        print(lower)
        lower = lower + 1
```

Recursive design

1. The loop's body continues execution while lower <= upper.

2. When the function executes, lower is incremented by 1, but upper never changes.

- The equivalent recursive function performs similar primitive operations, but the loop is replaced with a selection statement, and the assignment statement is replaced with a recursive call of the function.
- Here is the code with these changes:

Recursive design

```
def displayRange(lower, upper):
    """Outputs the numbers from lower through upper."""
    if lower <= upper:
        print(lower)
        displayRange(lower + 1, upper)
```


- Most recursive functions expect at least one argument.
- This data value is used to test for the base case that ends the recursive process, and it is modified in some way before each recursive step.
- The modification of the data value should produce a new data value that allows the function to reach the base case eventually..

Example 2

- The summation function computes and returns the sum of the numbers between lower and upper.
- In the recursive version, summation returns 0 if lower exceeds upper (the base case).
- Otherwise, the function adds lower to the summation of lower + 1 and upper and returns this result.

Cntd:

```
def summation(lower, upper):
    """Returns the sum of the numbers from lower through
    upper."""
    if lower > upper:
        return 0
    else:
        return lower + summation (lower + 1, upper)
```

Computing Factorial (!)

- 4! = 4 * 3 * 2 * 1 = 24
- N! = N * (N - 1) * (N - 2) * … * 1
- *Recursive definition* of factorial:
  - N! = 1, when N = 1
  - N! = N * (N - 1)!, otherwise

Define `factorial` Recursively

```
# N! = 1, when N = 1
# N! = N * (N - 1)!, otherwise

def factorial(n):
    if n == 1:
        return 1
    else:
        return n * factorial(n - 1)
```

What is the base case?

What is the recursive step?

Does the recursive step advance the process toward the base case?

Tracing `factorial`

```
# N! = 1, when N = 1
# N! = N * (N - 1)!, otherwise

def factorial(n):
    if n == 1:
        return 1
    else:
        return n * factorial(n - 1)
```

```
>>> factorial(4)          # With a trace of the process
n = 4
    n = 3
        n = 2
            n = 1
            factorial(1) = 1
        factorial(2) = 2
    factorial(3) = 6
factorial(4) = 24
```

Tracing Summation

```
def summation(lower, upper, margin):
    """Returns the sum of the numbers from lower through
upper,
    and outputs a trace of the arguments and return values
    on each call"""
    blanks = " " * margin
    print(blanks, lower, upper)
    if lower > upper:
        print(blanks, 0)
        return 0
    else:
        result = lower + summation(lower + 1, upper,
                                   margin + 4)
```

Tracing Summation

```
        print(blanks, result)
        return result


>>> summation (1, 4, 0)
1 4
    2 4
        3 4
            4 4
                5 4
                0
            4
        7
    9
10
10
```

Cntd:

- The displayed pairs of arguments are indented further to the right as the calls of summation proceed.
- Note that the value of lower increases by 1 on each call, whereas the value of upper stays the same.
- The final call of summation returns 0. As the recursion unwinds, each value returned is aligned with the arguments above it and increases by the current value of lower.

Recursive Defnitions

- Recursive functions are frequently used to design algorithms for computing values that have a recursive definition.
- A recursive definition consists of equations that state what a value is for one or more base cases and one or more recursive cases
- More formally, a recursive definition of the nth Fibonacci number is the following:
- $Fib(n) = 1$, when $n = 1$ or $n = 2$
- $Fib(n) = Fib(n - 1) + Fib(n - 2)$, for all $n > 2$

Fibonacci series

```
def fib(n):
    """Returns the nth Fibonacci number."""
    if n < 3:
        return 1
    else:
        return fib(n - 1) + fib(n - 2)
```

Infinite recursion

- Recursive functions tend to be simpler than the corresponding loops, but they still require thorough testing.
- One design error that might trip up a programmer occurs when the function can (theoretically) continue executing forever, a situation known as infinite recursion.
- Infinite recursion arises when the programmer fails to specify the base case or to reduce the size of the problem in a way that terminates the recursive process.

Example

```
>>> def runForever(n):
        if n > 0:
            runForever(n)
        else:
            runForever(n - 1)

>>> runForever(1)
Traceback (most recent call last):
  File "<pyshell#6>", line 1, in <module>
    runForever(1)
  File "<pyshell#5>", line 3, in runForever
    runForever(n)
  File "<pyshell#5>", line 3, in runForever
    runForever(n)
  File "<pyshell#5>", line 3, in runForever
    runForever(n)
  [Previous line repeated 989 more times]
  File "<pyshell#5>", line 2, in runForever
    if n > 0:
RecursionError: maximum recursion depth exceeded in comparison
```

The PVM keeps calling **`runForever(1)`** until there is no memory left to support another recursive call. Unlike an infinite loop, an infinite recursion eventually halts execution with an error message.

Cost and benefits of recursion

- At program startup, the PVM reserves an area of memory named a **call stack**.
- For each call of a function, recursive or otherwise, the PVM must allocate on the call stack a small chunk of memory called a **stack frame**.
- The system places the values of the arguments and the return address for each function call.
- Space for the function call's return value is also reserved in its stack frame.
- When a call returns or completes its execution, the return address is used to locate the next instruction in the caller's code, and the memory for the stack frame is deallocated.

EXAMPLE

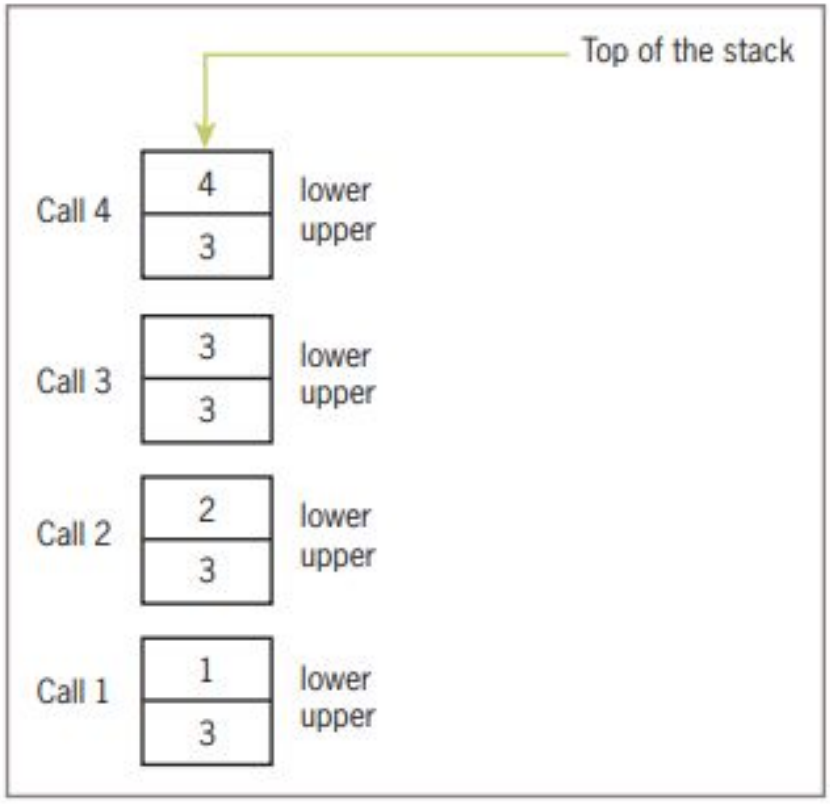


Figure 6-4 The stack frames for **`displayRange(1, 3)`**

What Is the Namespace?

- The *namespace* of a program has to do with the arrangement of names in a program
- These names refer to modules, data values, functions, methods, and data types
- Best to minimize the number of names in a program, but still have to organize!

Example Names

- Modules – **`math`**, **`random`**, **`doctor`**, **`generator`**
- Data types – **`int`**, **`float`**, **`str`**, **`list`**
- Variables – **`average`**, **`qualifiers`**, **`replacements`**
- Functions – **`print`**, **`input`**, **`sum`**, **`math.sqrt`**, **`reply`**
- Methods – **`append`**, **`split`**, **`join`**, **`lower`**

Properties of a Name

- *Binding time* - the point at which a name is defined to have a value
- *Scope* - the area of program text in which a name has a particular value
- *Lifetime* - the span of time during which a name has a particular value

```
replacements = {"I":"you", "me":"you", "my":"my""your",
                "we":"you", "us":"you", "mine":"yours"}

def changePerson(sentence):
    """Replaces first person pronouns with second person
    pronouns."""
    words = sentence.split( )
    replyWords = []
    for word in words:
        replyWords.append(replacements.get(word, word))
    return " ".join(replyWords)
```

Module variables

- Module variables. The names replacements and changePerson are introduced at the level of the module.
- Although replacements names a dictionary and changePerson names a function, they are both considered variables.

Parameters

- Parameters. The name sentence is a parameter of the function changePerson.
- A parameter name behaves like a variable and is introduced in a function or method header.
- The parameter does not receive a value until the function is called

Temporary variables

- Temporary variables. The names words, replyWords, and word are introduced in the body of the function changePerson.
- Like module variables, temporary variables receive their values as soon as they are introduced.

Method names

- Method names. The names split and join are introduced or defined in the str type.
- As mentioned earlier, a method reference always uses an object, in this case, a string, followed by a dot and the method name

The Scope of a Name

- A name's *scope* is the area of program text within which it has a particular value
- This value will be visible to some parts of a program but not to others
- Allows a single name to have different meanings in different contexts

The Scope of a Module Variable

```
qualifiers = ['Why do you say that ',
              'You seem to think that ',
              'Did I just hear you say that ']

for q in qualifiers: print(q)

def reply(sentence):
    return random.choice(qualifiers) + changePerson(sentence)
```

A *module variable* is visible throughout the module, even within a function definition

Module variables may be reset with assignment, but not within a function definition

The Scope of a Parameter

```
def reply(sentence):
    return random.choice(qualifiers) + changePerson(sentence)

def changePerson(sentence):
    sentence = sentence.lower()
    oldlist = sentence.split()
    newlist = []
    for word in oldlist:
        newlist.append(replacements.get(word, word))
    return " ".join(newlist)
```

A *parameter* is visible throughout its function definition but not outside of it

Parameters may be reset with assignment

The Scope of a Temporary Variable

```
def mySum(lyst):
    total = 0
    for number in lyst:
        total += number
    return total

def average(lyst):
    total = mySum(lyst)
    return total / len(lyst)
```

A *temporary variable* is visible within its function definition but not outside of it

Temporaries may be reset with assignment

The Scope of a Temporary Variable

```
def changePerson(sentence):
    oldlist = sentence.split()
    newlist = []
    for word in oldlist:
        newlist.append(replacements.get(word, word))
    return " ".join(newlist)
```

A *temporary variable* is visible throughout its function definition but not outside of it

Temporaries may be reset with assignment

The Scope of a Loop Variable

```
def changePerson(sentence):
    oldlist = sentence.split()
    newlist = []
    for word in oldlist:
        newlist.append(replacements.get(word, word))
    return " ".join(newlist)
```

A *loop variable* is like a temporary variable but is visible only in the body of the loop

Don't *ever* reset loop variables with assignment!

How Scope Works

```
x = "module"

def f():
    x = "temporary"
    print(x)

print(x)      # Outputs module
f()           # Outputs temporary
print(x)      # Outputs module
```

A temporary variable and a module variable can have the same name, but they refer to different storage spaces containing possibly different values

The Lifetime of a Variable

- A variable's lifetime is the period of time during program execution when the variable has memory storage associated with it.
- When a variable comes into existence, storage is allocated for it; when it goes out of existence, storage is reclaimed by the PVM
- The *lifetime* of a module variable is the span of time during which the module is active
- The *lifetime* of a parameter or a temporary variable is the span of time during which the enclosing function call is active

Default and Optional Parameters

One or more parameters can have default values, so the caller can omit some arguments

```
>>> round(3.1416)          # Default precision is 0
3

>>> round(3.1416, 3)       # Override the default
3.142

>>> list(range(5))         # Default lower bound is 0, and
[0,1,2,3,4]                # default step value is 1

>>> list(range(1, 5))      # Override the default lower bound
[1,2,3,4]

>>> list(range(1, 5, 2)) # Override lower bound and step
[1,3]
```

Keyword Parameters

Optional parameters can also be filled using keywords

```
>>> lyst = [4, 2, 6, 3]          # Items in random order

>>> lyst.sort()                  # Default sort of list

>>> lyst                         # Ascending order
[2,3,4,6]

>>> lyst = [4, 2, 6, 3]          # Reset to original order

>>> lyst.sort(reverse = True)    # Override default sort order

>>> lyst                         # Descending order
[6,4,3,2]
```

keyword arguments

- The **required arguments** are listed first in the function header. These are the ones that are “essential” for the use of the function by any caller.
- Following the required arguments are one or more **default arguments or keyword arguments**. These are assignments of values to the argument names.
- When the function is called without these arguments, their default values are automatically assigned to them.
- When the function is called with these arguments, the default values are overridden by the caller’s values.

Syntax

```
def <function name>(<required arguments>,
                    <key-1> = <val-1>, ... <key-n> = <val-n>)
```

Example

Here is an example of a function with one required argument and two default arguments and a session that shows these options:

```
>>> def example(required, option1 = 2, option2 = 3):
        print(required, option1, option2)

>>> example(1)                    # Use all the defaults
1 2 3
>>> example(1, 10)                # Override the first default
1 10 3
>>> example(1, 10, 20)            # Override all the defaults
1 10 20
>>> example(1, option2 = 20)      # Override the second default
1 2 20
```

Higher-Order Functions

- A *higher-order function* can receive another function as an argument
- The higher-order function then applies the argument function in some manner
- HOFs are a powerful way of simplifying code

Functions: first-class data objects.

- In Python, functions can be treated as first-class data objects.
- This means that they can be assigned to variables (as they are when they are defined), passed as arguments to other functions, returned as the values of other functions, and stored in data structures such as lists and dictionaries.

Example

```
>>> abs                              # See what abs looks like
<built-in function abs>
>>> import math
>>> math.sqrt
<built-in function sqrt>
>>> f = abs                          # f is an alias for abs
>>> f                                # Evaluate f
<built-in function abs>
>>> f(-4)                           # Apply f to an argument
4
>>> funcs = [abs, math.sqrt]        # Put the functions in a list
>>> funcs
[<built-in function abs>,   <built-in function sqrt>]
>>> funcs[1](2)                     # Apply math.sqrt to 2
1.4142135623730951
```

Mapping

- The first type of useful higher-order function to consider is called a mapping.
- This process applies a function to each value in a sequence (such as a list, a tuple, or a string) and returns a new sequence of the results.
- Suppose we have a list named words that contains strings that represent integers.
- We want to replace each string with the corresponding integer value.

Example

```
>>> words = ["231", "20", "-45", "99"]
>>> map(int, words)          # Convert all strings to ints
<map object at 0x14cbd90>
>>> words                    # Original list is not changed
['231', '20', '-45', '99']
>>> words = list(map(int, words))    # Reset variable to change it
>>> words
[231, 20, -45, 99]
```

Example: A List of Square Roots

```
oldlist = [2, 3, 4]

newlist = []

for n in oldlist:
    newlist.append(math.sqrt(n))
```

This type of operation is so common that Python includes a special function called **`map`** to simplify it:

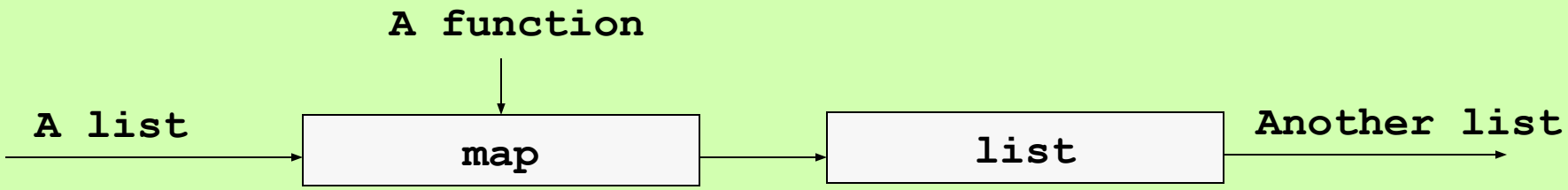
```
oldlist = [2, 3, 4]

newlist = list(map(math.sqrt, oldlist))
```

Note that **`map`** does not return a list, but we can run **`list`** to get one from it

Syntax of **map**

```
oldlist = [2, 3, 4]

newlist = list(map(math.sqrt, oldlist))
```

```
map(<a function>, <a list of arguments>)
```



Using **map**

```
def cube(n):
    return n ** 3

oldlist = [2, 3, 4]

newlist = list(map(cube, oldlist))

print(newlist)                          # Displays [8, 27, 64]
```

Define the function to use in the mapping, and then map it onto a list

Using **map**

```
oldlist = [2.17, 3.46, 4.54]

newlist = list(map(round, oldlist))

print(newlist)                          # Displays [2, 3, 5]
```

How could we round to 1 place of precision?

filtering

- A second type of higher-order function is called a filtering.
- In this process, a function called a predicate is applied to each value in a list.
- If the predicate returns True, the value passes the test and is added to a filter object (similar to a map object).
- Otherwise, the value is dropped from consideration

Example

```
>>> def odd(n): return n % 2 == 1
>>> list(filter(odd, range(10)))
[1, 3, 5, 7, 9]
```

Reducers

- Here we take a list of values and repeatedly apply a function to accumulate a single data value.
- A summation is a good example of this process.
- The first value is added to the second value, then the sum is added to the third value, and so on, until the sum of all the values is produced

Example

- The Python functools module includes a reduce function that expects a function of two arguments and a list of values

```
>>> from functools import reduce
>>> def add(x, y): return x + y
>>> def multiply(x, y): return x * y
>>> data = [1, 2, 3, 4]
>>> reduce(add, data)
10
>>> reduce(multiply, data)
24
```

lambda

- A lambda is an anonymous function. It has no name of its own, but it contains the names of its arguments as well as a single expression. When the lambda is applied to its arguments, its expression is evaluated, and its value is returned.

The syntax of a **lambda** is very tight and restrictive:

```
lambda <argname-1, ..., argname-n>: <expression>
```

Examples

```
>>> data = [1, 2, 3, 4]
>>> reduce(lambda x, y: x + y, data)     # Produce the sum
10
>>> reduce(lambda x, y: x * y, data)     # Produce the product
24
```

```
def summation(lower, upper):
    """Returns the sum of the numbers from lower
    through upper."""
    if lower > upper:
        return 0
    else:
        return reduce(lambda x, y: x + y,
                      range(lower, upper + 1))
```

jump table

- A jump table is a dictionary of functions keyed by command names.
- At program startup, the functions are defined and then the jump table is loaded with the command names and their associated functions.
- The function runCommand uses its command argument to look up the function in the jump table and then calls this function.
- Here is the modified version of runCommand:

```
def runCommand(command):     # How simple can it get?
    jumpTable[command]()
```

Note that this function makes two important simplifying assumptions: the command string is a key in the jump table, and its associated function expects no arguments.

Let's assume that the functions **insert**, **replace**, and **remove** are keyed to the commands **'1'**, **'2'**, and **'3'**, respectively. Then the setup of the jump table is straightforward:

```
# The functions named insert, replace, and remove
# are defined earlier
jumpTable = {}
jumpTable['1'] = insert
jumpTable['2'] = replace
jumpTable['3'] = remove
```

Exercises

- 1. Write the code for a mapping that generates a list of the absolute values of the num bers in a list named numbers.
- 2. Write the code for a filtering that generates a list of the positive numbers in a list named numbers. You should use a lambda to create the auxiliary function.
- 3. Write the code for a reducing that creates a single string from a list of strings named words

END