The Practice of Computing Using PYTHON

William Punch Richard Enbody

Chapter 4

Working with Strings

Sequence of Characters

- We’ve talked about strings being a sequence of characters.
- A string is indicated between ‘ ‘ or “ “
- The exact sequence of characters is maintained (for the most part).

And Then There is “““ ”””

- Triple quotes preserve both the vertical and horizontal formatting of the string
- Allow you to type tables, paragraphs, whatever and preserve the formatting

```
“““this is
a test
today”””
```

Strings

Can use single or double quotes:

- S = “spam”
- s = ‘spam’

Just don’t mix them!

- myStr = ‘hi mom” ⇒ ERROR

Inserting an apostrophe:

- A = “knight’s” # *mix up the quotes*
- B = ‘knight\’s’ # *escape* single quote

The Index

- Because the elements of a string are a sequence, we can associate each element with an **index**, a location in the sequence:
  - positive values count up from the left, beginning with index 0
  - negative values count down from the right, starting with -1

characters	H	e	l	l	o		W	o	r	l	d
index	0	1	2	3	4	5	6	7	8	9	10
									...	−2	−1

FIGURE 4.1 The index values for the string 'Hello World'.

Accessing an Element

- A particular element of the string is accessed by the index of the element surrounded by square brackets [ ]

helloStr = ‘Hello World’

print helloStr[1] => prints ‘e’

print helloStr[-1] => prints ‘d’

print helloStr[11] => ERROR

Slicing: the Rules

- slicing is the ability to select a subsequence of the overall sequence
- uses the syntax [start : finish], where:
  - start is the index of where we start the subsequence
  - finish is the index of **one after** where we end the subsequence
- if either start or finish are not provided, it defaults to the beginning of the sequence for start and the end of the sequence for finish

```
helloString[6:10]
```

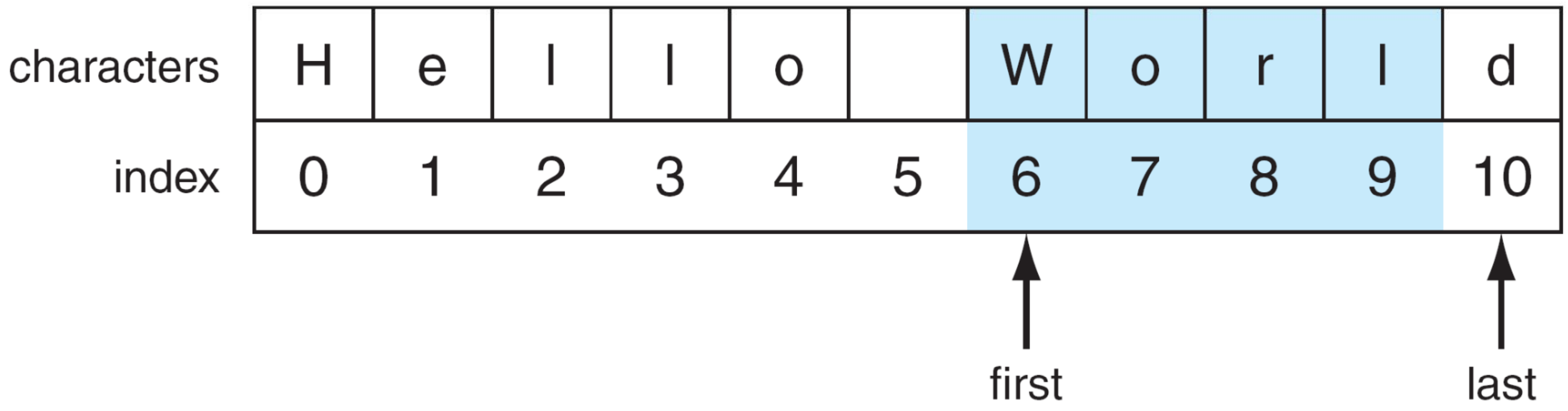


FIGURE 4.2 Indexing subsequences with slicing.

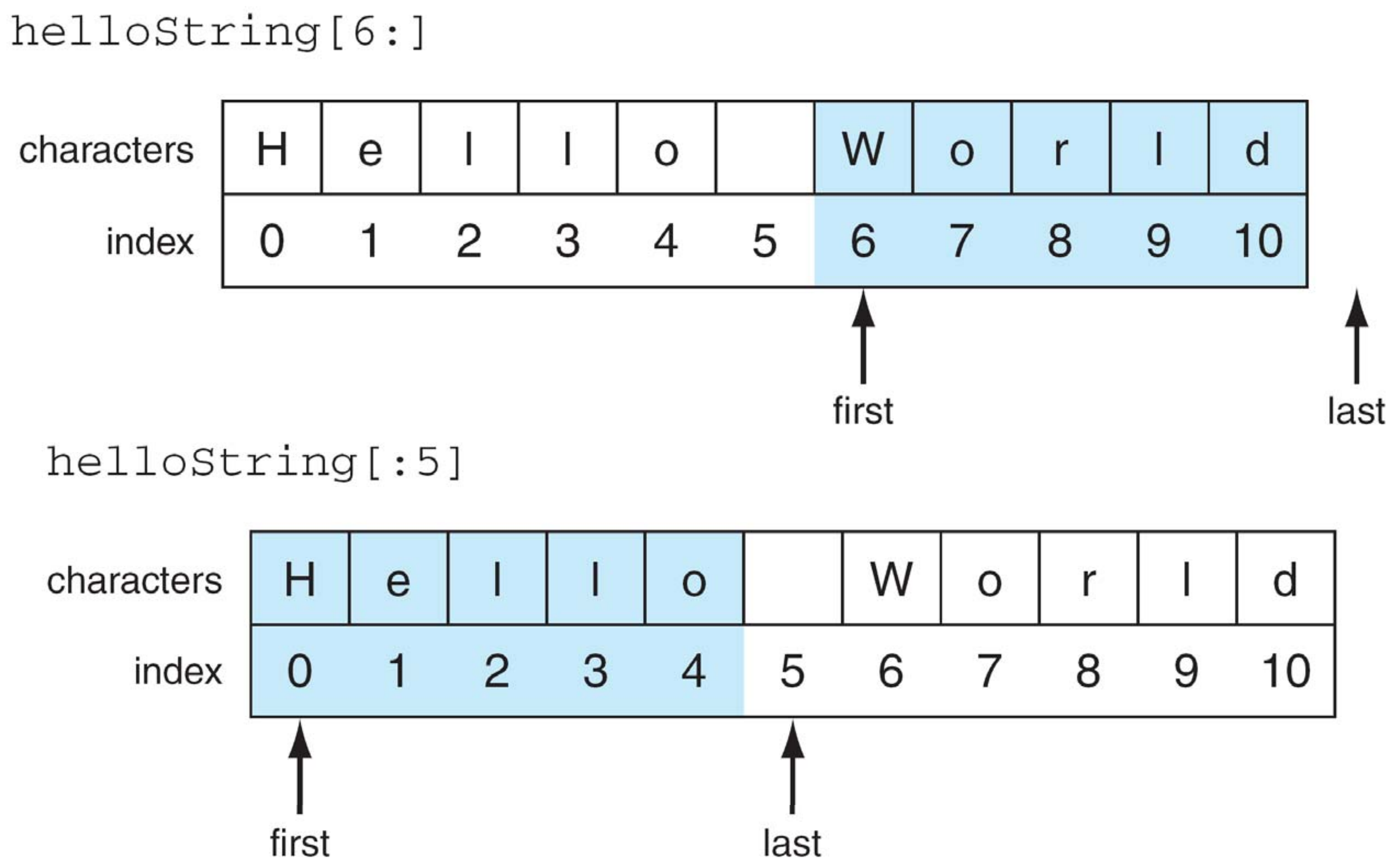


FIGURE 4.3 Two default slice examples.

```
helloString[3:-2]
```

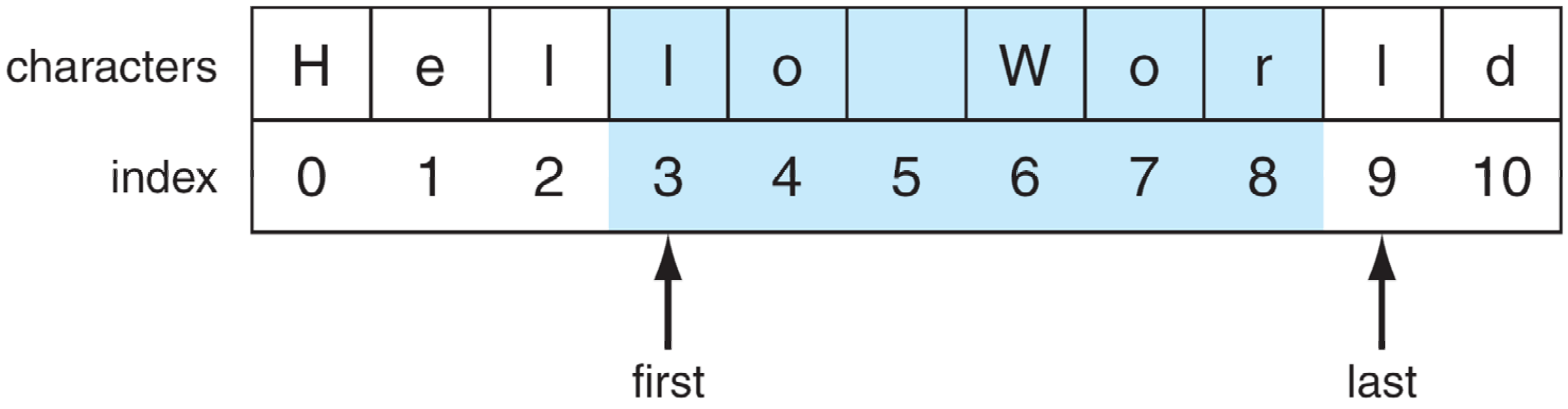


FIGURE 4.5 Another slice example.

Extended Slicing

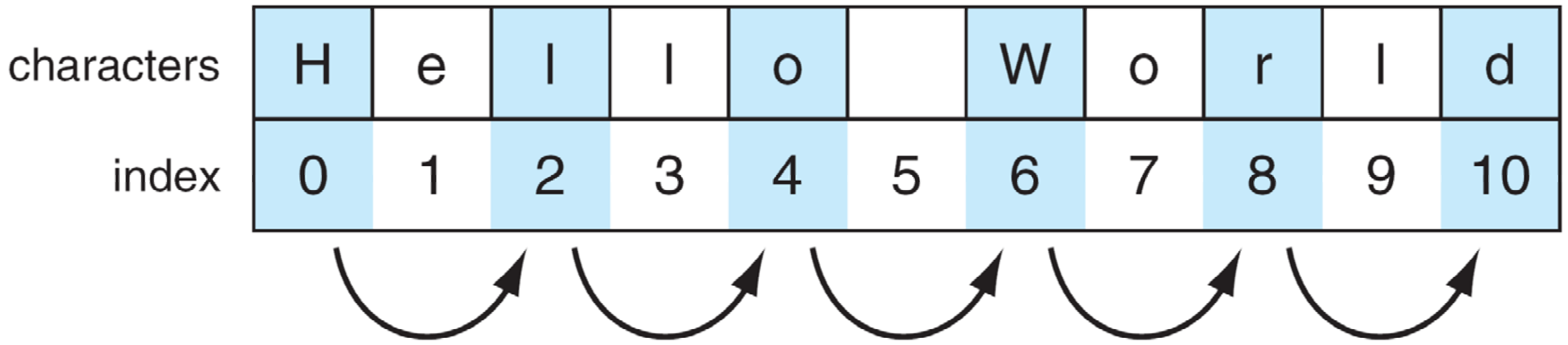
- also takes three arguments:
  - [start:finish:countBy]
- defaults are:
  - start is beginning, finish is end, countBy is 1

myStr = ‘hello world’

myStr[0:11:2] $\Rightarrow$ ‘hlowrd’

- every other letter

```
helloString[::2]
```



characters	H	e	l	l	o		W	o	r	l	d
index	0	1	2	3	4	5	6	7	8	9	10

FIGURE 4.6 Slicing with a step.

Some Python “Idioms”

- Idioms are python “phrases” that are used for a common task that might be less obvious to non-python folk.
- How to make a copy of a string:

```
myStr = ‘hi mom’
newStr = myStr[:]
```

- How to reverse a string:

```
myStr = ‘madam I’m adam’
reverseStr = myStr[::-1]
```

String Operations

Basic String Operations

s = ‘spam’

- length operator len()

len(s) $\Rightarrow$ 4

- + is concatenate

newStr = ‘spam’ + ‘-’ + ‘spam-’

print newStr $\Rightarrow$ `spam-spam-`

- * is repeat, the number is how many times

newStr * 3 $\Rightarrow$

spam-spam-spam-spam-spam-spam-

Some Details

- Both + and * on strings make a new string, but does not modify the arguments.
- Order of operation is important for concatenation.
- The types required are specific. For concatenation you need two strings; for repetition, a string and an integer.

What Does A + B Mean?

- What operation does the above represent? It depends on the types!
  - – two strings, concatenation
  - – two integers addition
- The operator + is **overloaded**.
  - – the operation + performs depends on the types it is working on

String Comparisons, Single Char

- ASCII takes the English letters, numbers and punctuation marks and associates them with an integer number
- Single character comparisons are based on that number

Dec	Hx	Oct	Char		Dec	Hx	Oct	Html	Chr	Dec	Hx	Oct	Html	Chr	Dec	Hx	Oct	Html	Chr
0	0	000	NUL	(null)	32	20	040	&#32;	Space	64	40	100	&#64;	@	96	60	140	&#96;	`
1	1	001	SOH	(start of heading)	33	21	041	&#33;	!	65	41	101	&#65;	A	97	61	141	&#97;	a
2	2	002	STX	(start of text)	34	22	042	&#34;	"	66	42	102	&#66;	B	98	62	142	&#98;	b
3	3	003	ETX	(end of text)	35	23	043	&#35;	#	67	43	103	&#67;	C	99	63	143	&#99;	c
4	4	004	EOT	(end of transmission)	36	24	044	&#36;	$	68	44	104	&#68;	D	100	64	144	&#100;	d
5	5	005	ENQ	(enquiry)	37	25	045	&#37;	%	69	45	105	&#69;	E	101	65	145	&#101;	e
6	6	006	ACK	(acknowledge)	38	26	046	&#38;	&	70	46	106	&#70;	F	102	66	146	&#102;	f
7	7	007	BEL	(bell)	39	27	047	&#39;	'	71	47	107	&#71;	G	103	67	147	&#103;	g
8	8	010	BS	(backspace)	40	28	050	&#40;	(	72	48	110	&#72;	H	104	68	150	&#104;	h
9	9	011	TAB	(horizontal tab)	41	29	051	&#41;	)	73	49	111	&#73;	I	105	69	151	&#105;	i
10	A	012	LF	(NL line feed, new line)	42	2A	052	&#42;	*	74	4A	112	&#74;	J	106	6A	152	&#106;	j
11	B	013	VT	(vertical tab)	43	2B	053	&#43;	+	75	4B	113	&#75;	K	107	6B	153	&#107;	k
12	C	014	FF	(NP form feed, new page)	44	2C	054	&#44;	,	76	4C	114	&#76;	L	108	6C	154	&#108;	l
13	D	015	CR	(carriage return)	45	2D	055	&#45;	-	77	4D	115	&#77;	M	109	6D	155	&#109;	m
14	E	016	SO	(shift out)	46	2E	056	&#46;	.	78	4E	116	&#78;	N	110	6E	156	&#110;	n
15	F	017	SI	(shift in)	47	2F	057	&#47;	/	79	4F	117	&#79;	O	111	6F	157	&#111;	o
16	10	020	DLE	(data link escape)	48	30	060	&#48;	0	80	50	120	&#80;	P	112	70	160	&#112;	p
17	11	021	DC1	(device control 1)	49	31	061	&#49;	1	81	51	121	&#81;	Q	113	71	161	&#113;	q
18	12	022	DC2	(device control 2)	50	32	062	&#50;	2	82	52	122	&#82;	R	114	72	162	&#114;	r
19	13	023	DC3	(device control 3)	51	33	063	&#51;	3	83	53	123	&#83;	S	115	73	163	&#115;	s
20	14	024	DC4	(device control 4)	52	34	064	&#52;	4	84	54	124	&#84;	T	116	74	164	&#116;	t
21	15	025	NAK	(negative acknowledge)	53	35	065	&#53;	5	85	55	125	&#85;	U	117	75	165	&#117;	u
22	16	026	SYN	(synchronous idle)	54	36	066	&#54;	6	86	56	126	&#86;	V	118	76	166	&#118;	v
23	17	027	ETB	(end of trans. block)	55	37	067	&#55;	7	87	57	127	&#87;	W	119	77	167	&#119;	w
24	18	030	CAN	(cancel)	56	38	070	&#56;	8	88	58	130	&#88;	X	120	78	170	&#120;	x
25	19	031	EM	(end of medium)	57	39	071	&#57;	9	89	59	131	&#89;	Y	121	79	171	&#121;	y
26	1A	032	SUB	(substitute)	58	3A	072	&#58;	:	90	5A	132	&#90;	Z	122	7A	172	&#122;	z
27	1B	033	ESC	(escape)	59	3B	073	&#59;	;	91	5B	133	&#91;	[	123	7B	173	&#123;	{
28	1C	034	FS	(file separator)	60	3C	074	&#60;	<	92	5C	134	&#92;	\	124	7C	174	&#124;	|
29	1D	035	GS	(group separator)	61	3D	075	&#61;	=	93	5D	135	&#93;	]	125	7D	175	&#125;	}
30	1E	036	RS	(record separator)	62	3E	076	&#62;	>	94	5E	136	&#94;	^	126	7E	176	&#126;	~
31	1F	037	US	(unit separator)	63	3F	077	&#63;	?	95	5F	137	&#95;	_	127	7F	177	&#127;	DEL

Source: www.asciitable.com

Comparisons Within Sequence

- It makes sense to compare within a sequence (lower case, upper case, digits).
  - ‘a’ < ‘b’ True
  - ‘A’ < ‘B’ True
  - ‘1’ < ‘9’ True
- Can be weird outside of the sequence:
  - ‘a’ < ‘A’ False
  - ‘a’ < ‘0’ False

Whole Strings

- Compare the first element of each string:
  - if they are equal, move on to the next character in each
  - if they are not equal, the relationship between those to characters are the relationship between the string
  - if one ends up being shorter (but equal), the shorter is smaller

Examples

- ‘a’ < ‘b’   True
- ‘aaab’ < ‘aaac’
  - First difference is at the last char. ‘b’<‘c’ so ‘aaab’ is less than ‘aaac’. True.
- ‘aa’ < ‘aaz’
  - The first string is the same but shorter. Thus it is “smaller”. True.

Membership Operations

- Can check to see if a substring exists in the string, the `in` operator. Returns True or False

```
myStr = ‘aabbccdd’
‘a’ in myStr ⇒ True
‘abb’ in myStr ⇒ True
‘x’ in myStr ⇒ False
```

Strings are Immutable

- Strings are immutable, that is you cannot change one once you make it:
  - aStr = ‘spam’
  - aStr[1] = ‘l’ $\Rightarrow$ ERROR
- However, you can use it to make another string (copy it, slice it, etc).
  - newStr = aStr[:1] + ‘l’ + aStr[2:]
  - aStr $\Rightarrow$ ‘spam’
  - newStr => ‘slam’

String Methods and Functions

String Method

- A **method** is a variation on a function
  - like a function, it encapsulates some computations
  - like a function, it has input arguments and an output
- Unlike a function, it is applied in the context of a particular object.
- This is indicated by the ‘dot notation’ invocation

Example

- upper is the name of a method. It generates a new string that has all upper case characters of the string it was called with.

myStr = ‘Python Rules!’

myStr.upper() $\Rightarrow$ ‘PYTHON RULES!’

- The string myStr called the upper() method, indicated by the dot between them.

More Dot Notation

- In generation, dot notation looks like:
  - object.method(…)
- It means that the object in front of the dot is calling a method that is associated with that object’s type.
- The methods that can be called are tied to the type of the object calling it. Each type has different methods.

Find Method

```
myStr = ‘hello’
myStr.find(‘l’)          # find index of ‘l’ in myStr
```

$\Rightarrow$ 2

Finds the first instance of the argument in the string object

Note how the method ‘find’ operates on the string object myStr and the two are associated by using the “dot” notation: myStr.find(‘l’).

Optional Arguments

Some methods have optional arguments:

- if the user doesn't provide one of these, a default is assumed
- find has a default second argument of 0, where the search begins

aStr = 'He had the bat'

aStr.find('t') $\Rightarrow$ 7 # 1st 't',start @ 0

aStr.find('t',8) $\Rightarrow$ 13 # 2nd 't'

Nesting Methods

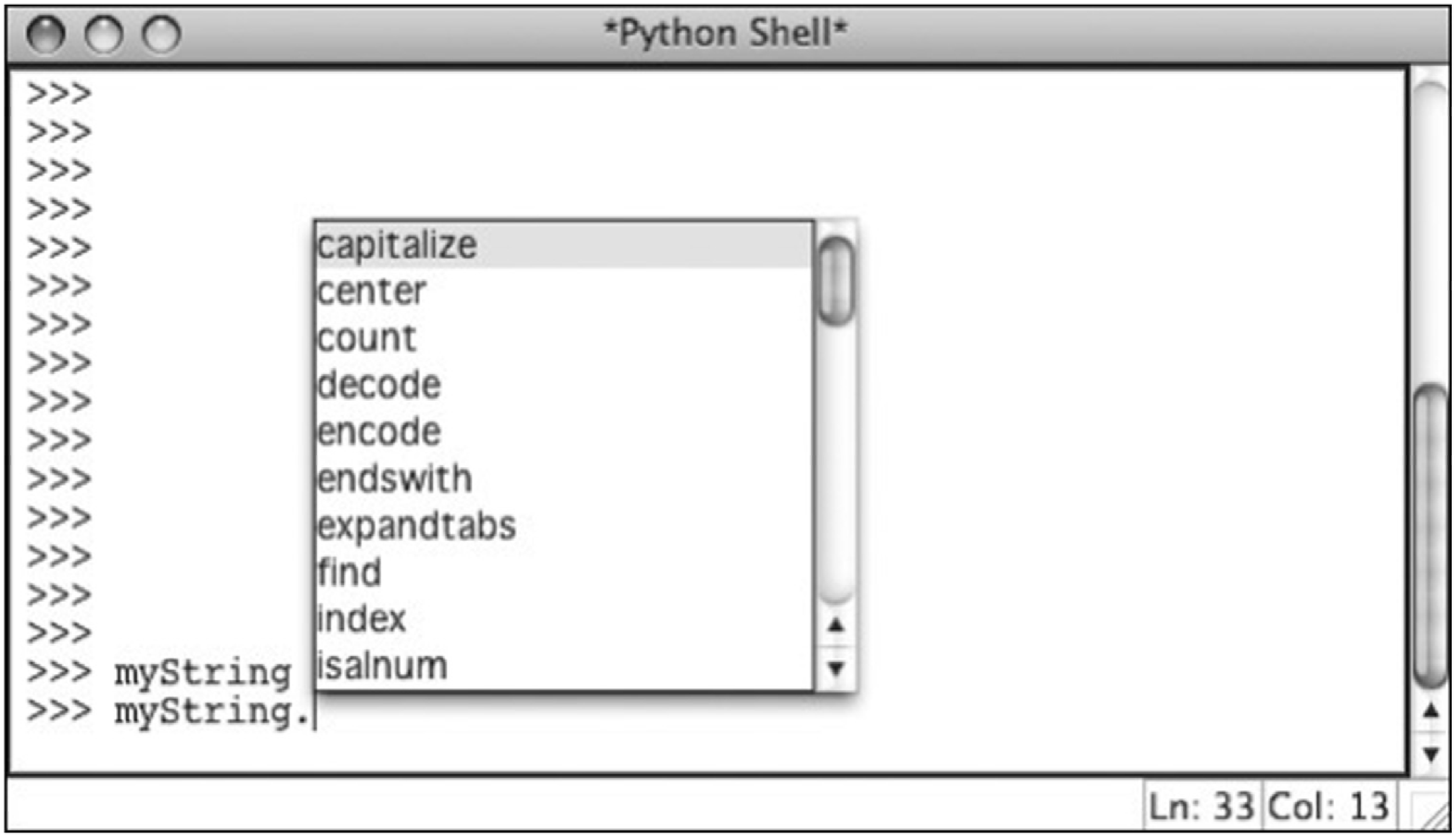
- You can “nest” methods, that is, the result of one method as an argument to another.
- Remember that parentheses expressions are done “inside out”: do the inner parenthetical expression first, then the next, using the result as an argument.

```
aStr.find(‘t’, aStr.find(‘t’)+1)
```

- Translation: find the second ‘t’.

How to Know?

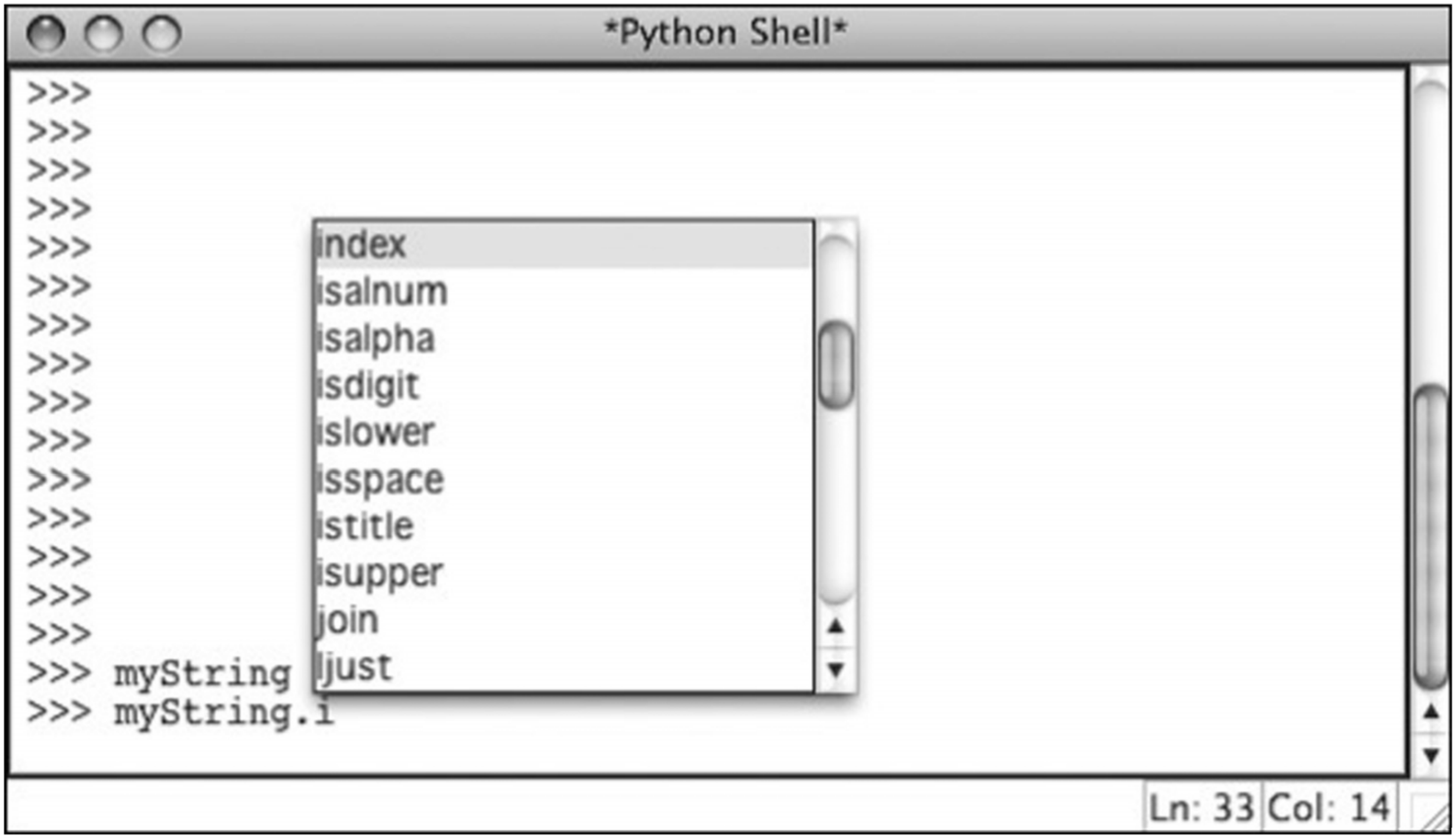
- You can use IDLE to find available methods for any type. You enter a variable of the type, followed by the ‘.’ (dot) and then a tab (also Python documentation).
- Remember, methods match with a type. Different types have different methods.
- If you type a method name, IDLE will remind you of the needed and optional arguments.



```
>>>
>>>
>>>
>>>
>>>
>>>
>>>
>>>
>>>
>>>
>>>
>>>
>>>
>>>
>>>
>>> myString
>>> myString.
```

```
capitalize
center
count
decode
encode
endswith
expandtabs
find
index
isalnum
```

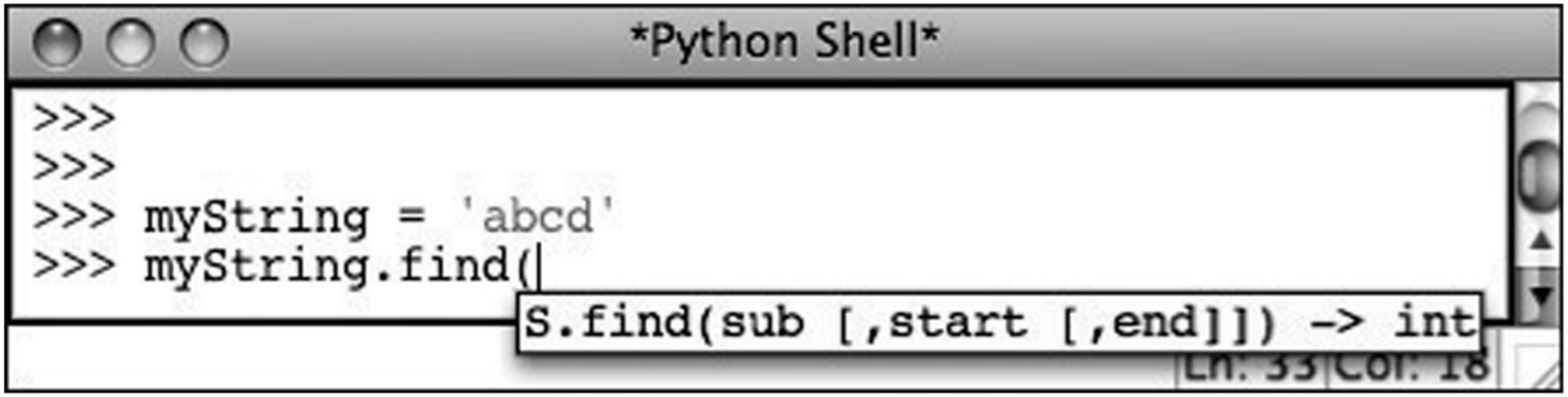
FIGURE 4.7 In IDLE, tab lists potential methods.



```
>>>
>>>
>>>
>>>
>>>
>>>
>>>
>>>
>>>
>>>
>>>
>>>
>>>
>>>
>>>
>>> myString
>>> myString.i
```

```
index
isalnum
isalpha
isdigit
islower
isspace
istitle
isupper
join
ljust
```

FIGURE 4.8 In IDLE, tab lists potential methods, with leading letter.



```
>>>
>>>
>>> myString = 'abcd'
>>> myString.find(
S.find(sub [,start [,end]]) -> int
```

FIGURE 4.9 IDLE pop-up provides help with function arguments and return types.

More Methods

(Even more exist: http://docs.python.org/lib/string-methods.html)

- s.capitalize
- s.center(width)
- s.count(sub,[,start [,end]])
- s.ljust(width)
- s.lower()
- s.upper()
- s.lstrip()
- s.rfind(sub, [,start [,end]])
- s.splitlines([keepends])
- s.strip()
- s.translate(table [, delchars])

String Formatting

String Formatting, Better Printing

- So far, we have just used the defaults of the print function.
- We can do many more complicated things to make that output “prettier” and more pleasing.
- We will apply it to our “display” function.

Basic Form

- To understand string formatting, it is probably best to start with an example:

```
print “Sorry, is this the %d minute %s?” % (5,
   ‘ARGUMENT’ )
```

prints Sorry, is this the 5 minute ARGUMENT

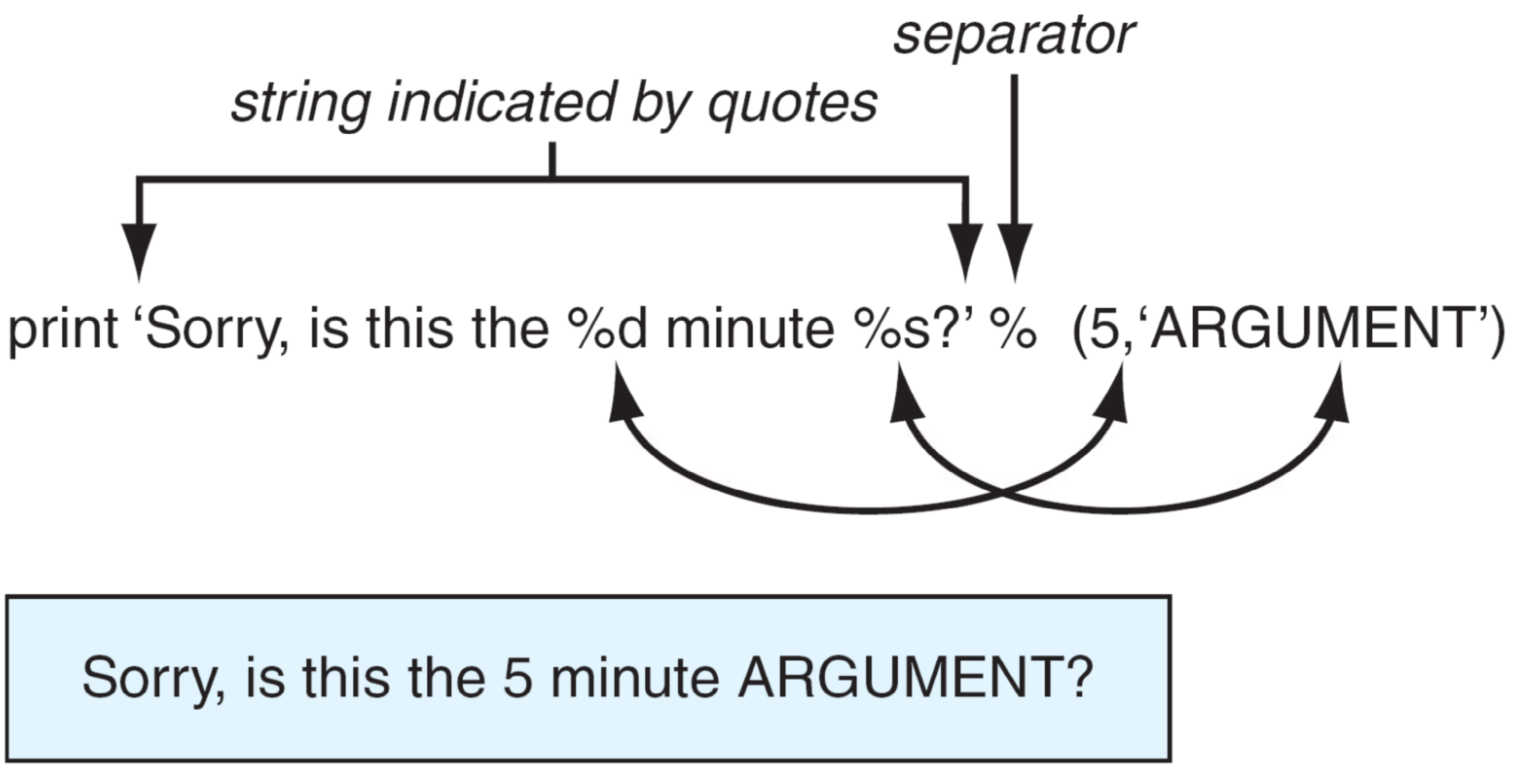
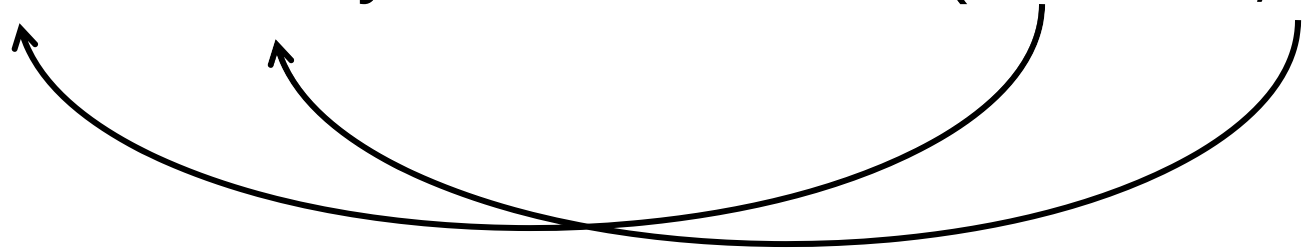

separator
string indicated by quotes
print ‘Sorry, is this the %d minute %s?’ % (5,‘ARGUMENT’)
Sorry, is this the 5 minute ARGUMENT?

FIGURE 4.10 String formatting example.

Matching Object to Descriptor

- Objects are matched in order with format descriptors. The substitution is made and resulting string printed

```
print “%s is %d years old” % (“Bill”,25)
```



prints `Bill is 25 years old`

Many Descriptors

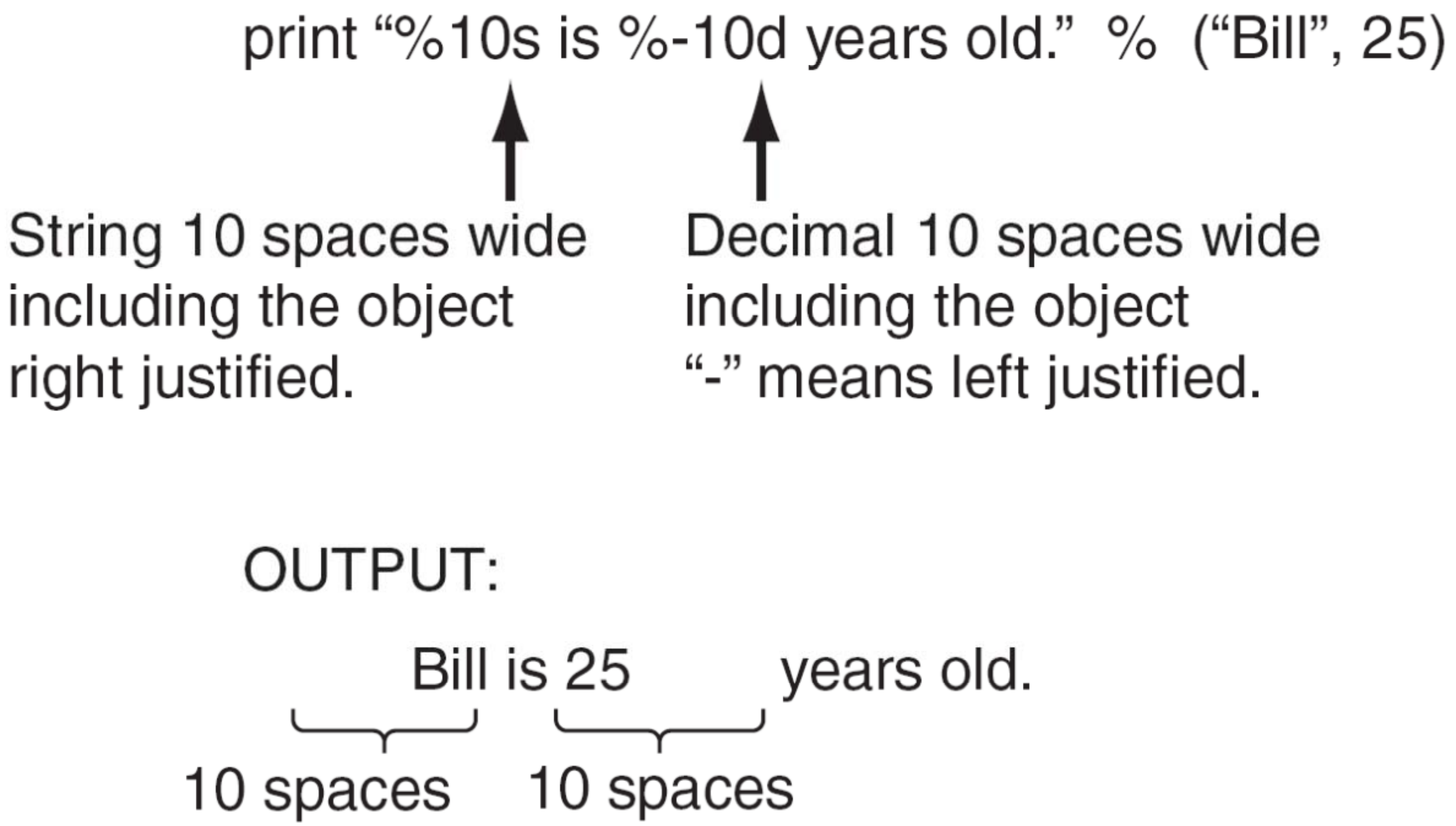
- %s string
- %d decimal
- %e floating point exponent
- %f floating point decimal
- %u unsigned integer
- and others

Format String

- The format string contains a set of format descriptors that describe how an object is to be printed.
- Overall:

%[width][.precision]code

where [ ] are optional



```
print "%10s is %-10d years old." % ("Bill", 25)
```

String 10 spaces wide including the object right justified.
Decimal 10 spaces wide including the object "-" means left justified.
OUTPUT:
10 spaces
10 spaces

```
      Bill is 25        years old.
```

FIGURE 4.11 String formatting with width descriptors.

Precision

- print math.pi
  - 3.14159265359
- print “%.4f” % math.pi
  - `3.1416` (4 decimal points of precision, with rounding)
- print “%10.2f” % math.pi
  - `      3.14` (10 spaces total including the number and the decimal point

Iteration

Iteration Through a Sequence

- To date, we have seen the while loop as a way to iterate over a suite (a group of python statements)
- We briefly touched on the for statement for iteration, such as the elements of a list or a string

`for` Statement

We use the for statement to process each element of a list, one element at a time:

```
for item in sequence:
    suite
```

What `for` means

```
myStr='abc'
for myVar in 'abc':
        print myVar
```

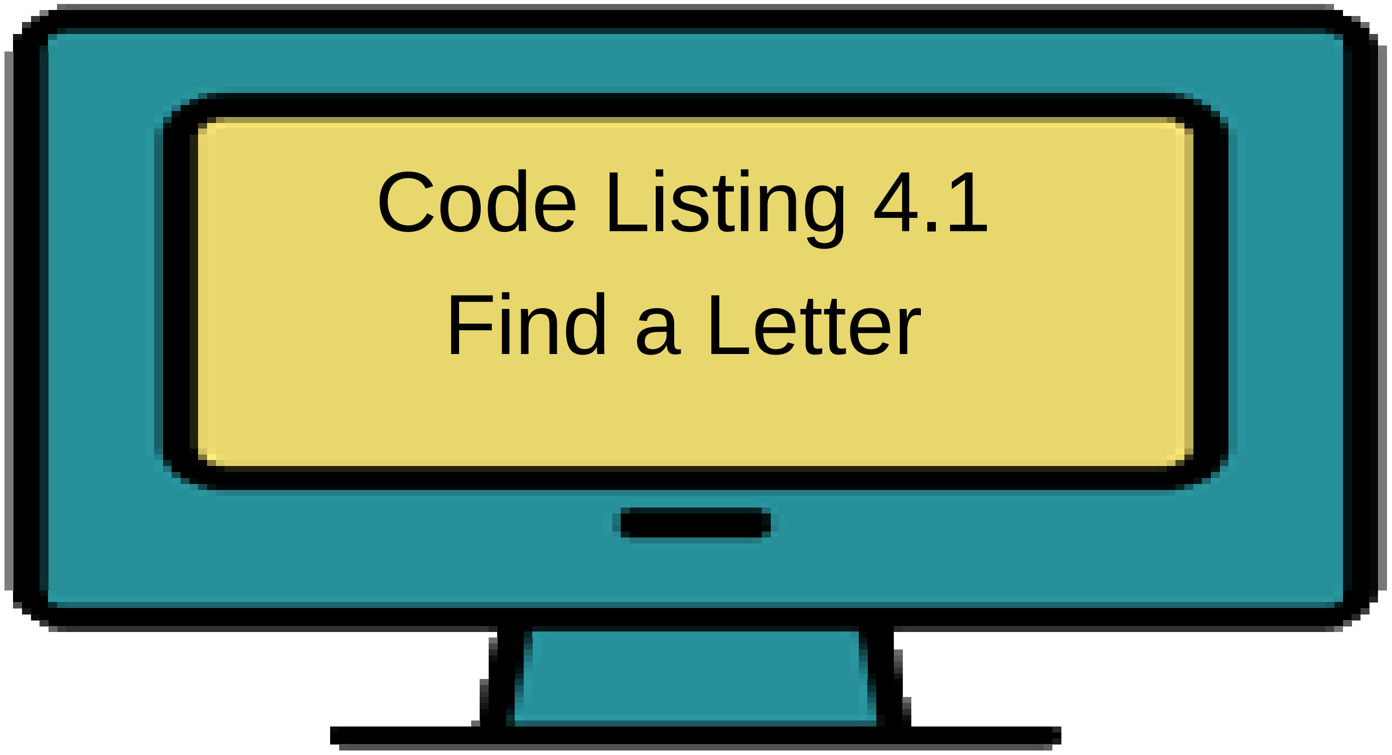
- first time through, myVar='a' (myStr[0])
- second time through, myVar='b' (myStr[1])
- third time through, myVar='c' (myStr[2])
- no more sequence left, we quit

Power of the for Statement

- Sequence iteration as provided by the for statement is very powerful and very useful in Python.
- Allows you to write some very “short” programs that do powerful things.

Example: Finding a letter

- How might we find the index of a given letter in the absence of a find() method?



Code Listing 4.1
Find a Letter

```
river = 'Mississippi'
target = raw_input('Input character to find: ')
for index in range(len(river)): #for each index
    if river[index] == target:  #check
        print "Letter found at index: ", index
        break                      # stop searching
else:
    print 'Letter‘, target, 'not found in', river
```

Example: Finding all letters

- How might we modify the program to print ALL indices of a given letter?

Split Method

- The split method will take a string and break it into multiple new string parts depending on what the argument character is.
- If no argument is provided, split is based on any whitespace character (tab, blank, etc).
- You can assign the pieces with multiple assignment if you know how many pieces are yielded.

Reorder a Name

```
origName = ‘John Marwood Cleese’
first, mid, last = origName.split()
name = last + ‘, ‘ + first + ‘ ‘ + mid
print name
```

Strip Method

- The strip function will return a copy of the string with the specified characters removed from the beginning and end of the string.
- If no argument is provided, strip is based on any whitespace character (tab, blank, etc).
- Example:

```
s = '     Hello World     '
print s.strip()
Hello World
```

Replace method

- The replace method will return a copy of the string with all instances of the first argument replaced with the second argument.
- Example:

```
s = 'hello world'
print s.replace('l', 'b')
hebbo worbd
```

Palindromes and the Rules

- A palindrome is a string that prints the same forward and backwards
- Same implies that:
  - case does not matter
  - punctuation is ignored
- “Madam I’m Adam” is thus a palindrome

Example: Palindromes

- How might we determine if text inputted by the user is a palindrome?

Lower Case and Punctuation

- Every letter is converted using the lower method
- Import string, brings in a series of predefined sequences (string.digits, string.punctuation, string.whitespace)
- We remove all non-wanted characters with the replace method. First arg is what to replace, the second the replacement.

Code Listing 4.4 Palindromes

```
# first part
import string
originalString = raw_input('Input a string: ')
modifiedStr = originalString.lower()
badChars = string.whitespace +
string.punctuation
for char in modifiedStr:
    if char in badChars:  # remove bad
        modifiedStr = modifiedStr.replace(char, '')
```

```
# second part
if modifiedStr == modifiedStr[::-1]: # pal ?
     print 'The original string is:  %s\n\
     the modified string is: %s\n\
     the reversal is:        %s\n\
    The string is a palindrome' %
    (originalString, modifiedStr, modifiedStr[::-1])
else:
    # similar printing for not a palindrome
```