LẬP TRÌNH VỚI PYTHON
(Programming with Python)

Th.S Nguyễn Hoàng Thành

Email: thanhnh@ptithcm.edu.vn

Tel: 0909 682 711

Strings

Strings

- Python string: an **ordered collection of characters** used to store and represent text-based information.
  - anything that can be encoded as text:
    - symbols and words,
    - contents of text files loaded into memory,
    - Internet addresses,
    - Python programs,…

Table 7.1 Common string literals and operations

Operations	Interpretation
S = ''	Empty String
S = "Spam's"	Double quotes, same as single
S = 's\np\ta\x00m'	Escape sequences
S = """.."""	Triple-quoted block strings
S = r'\temp\spam'	Raw strings
S = b'spam'	Byte strings in 3.0
S = u'spam'	Unicode strings in 2.6 only
S1 + S2	Concatenate
S * 3	Repeat

Table 7.1 Common string literals and operations (con't)

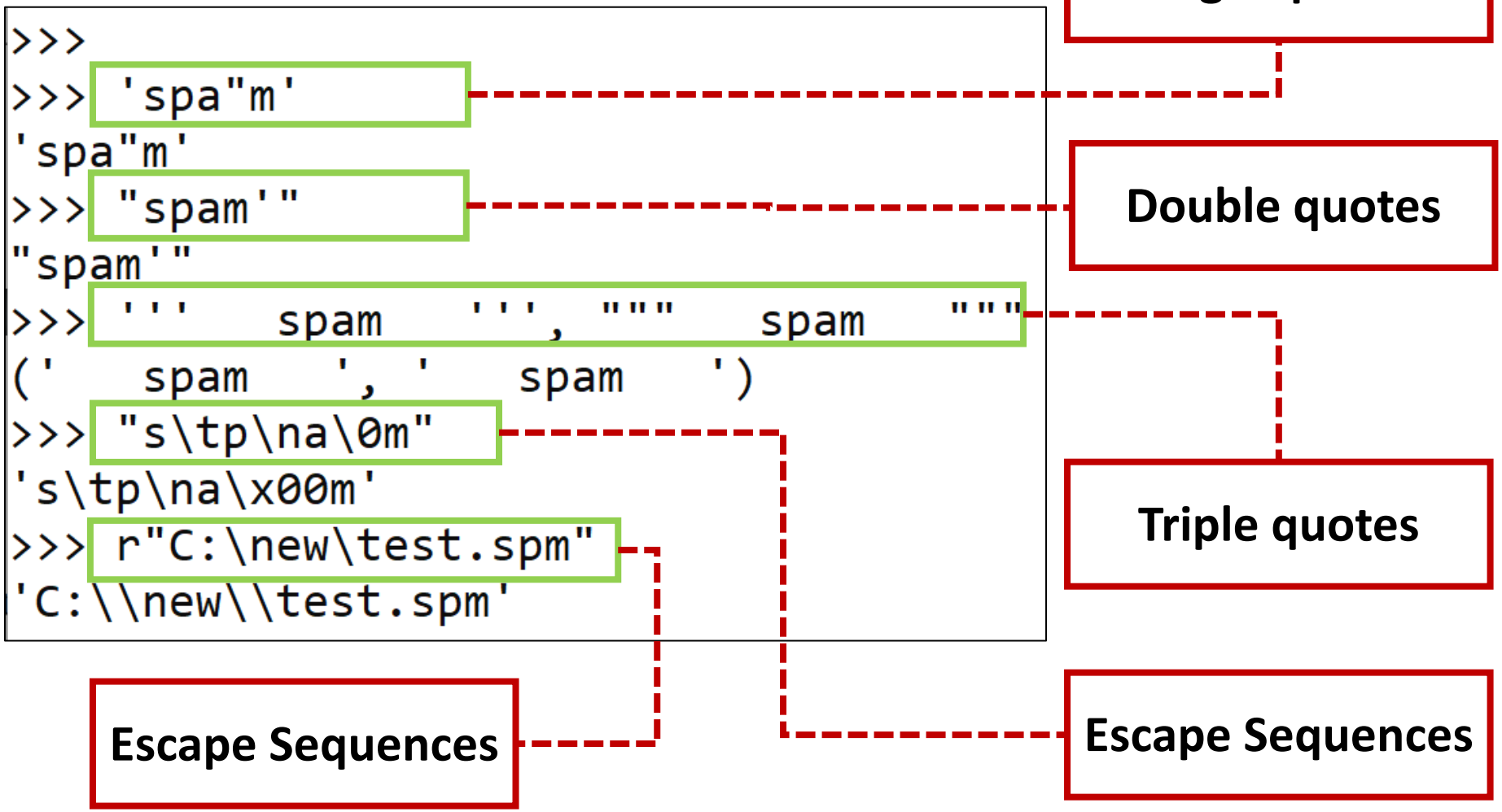
Operations	Interpretation
S[i]	Index
S[i:j]	Slice
len(S)	length
"a %s parrot" % kind	String formatting expression
"a {0} parrot".format(kind)	String formatting method in 2.6 and 3.0
S.find('pa')	String method calls: search,
S.rstrip()	remove whitespace,
S.replace('pa', 'xx')	replacement,
S.split(',')	split on delimiter,

Table 7.1 Common string literals and operations (con't)

Operations	Interpretation
S.isdigit()	content test,
S.lower()	case conversion,
S.endswith('spam')	end test,
'spam'.join(strlist)	delimiter join,
S.encode('latin-1')	Unicode encoding, etc.
for x in S: print(x)	Iteration
'spam' in S	membership
[c * 2 for c in S]	
map(ord, S)	

1. String Literals

- So **many ways** to write them in your code



1.1 Single- and Double-Quoted Strings Are the Same

- **Single** and **double quote** characters are **interchangeable**. So, can be write enclosed in either **two single** or two **double quotes**
- **embed** a single quote character in a string enclosed in double quote characters, and vice versa.
- **concatenates** adjacent string literals in any expression

```
>>> 'shrubbery', "shrubbery"
('shrubbery', 'shrubbery')
>>> 'knights"s', "knight's"
('knights"s', "knight's")
>>> title = "Meaning " 'of' " Life"
>>> title
'Meaning of Life'
```

1.2 Escape Sequences Represent Special Bytes

- **backslashes** are used to introduce special byte codings known as **escape sequences.**
- The character \, and **one or more** characters following are **replaced** with a **single character** in the resulting string object.

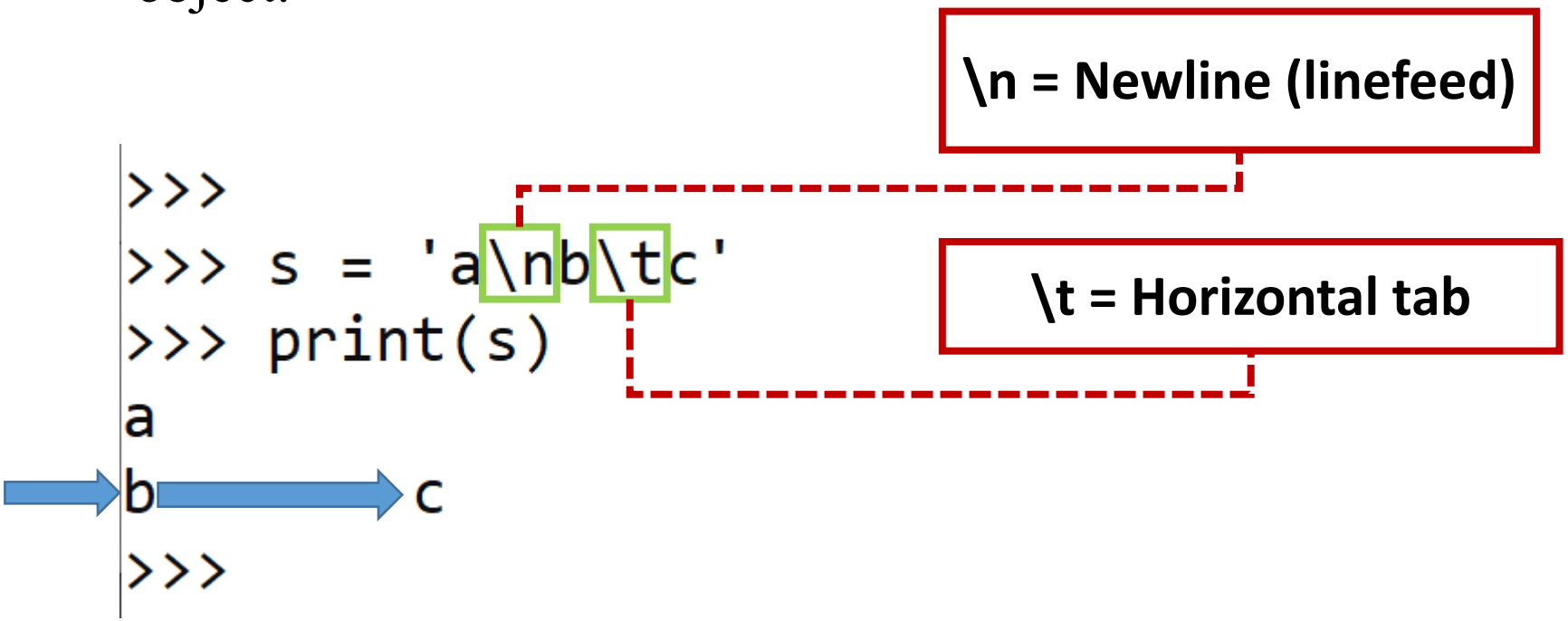


Table 7.2 String backslash characters

Escape	Meaning
\newline	Ignored (continuation line)
\\	Backslash (stores one \)
\'	Single quote (stores ')
\"	Double quote (stores ")
\a	Bell
\b	Backspace
\f	Formfeed
\n	Newline (linefeed)
\r	Carriage return

Table 7.2 String backslash characters (con't)

Escape	Meaning
\t	Horizontal tab
\v	Vertical tab
\xhh	Character with hex value hh (at most 2 digits)
\ooo	Character with octal value ooo (up to 3 digits)
\0	Null: binary 0 character (doesn't end string)
\N{ id }	Unicode database ID
\uhhhh	Unicode 16-bit hex
\Uhhhhhhhh	Unicode 32-bit hex
\other	Not an escape (keeps both \ and other)

1.3 Raw Strings Suppress Escapes

- Escape sequences are handy for **embedding special byte codes** within strings.
- But, **escapes can lead to trouble**

```
>>>
>>> myfile = open('C:\new\text.dat', 'w')
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
OSError: [Errno 22] Invalid argument: 'C:\new\text.dat'
```

\n and \t

1.3 Raw Strings Suppress Escapes (con't)

- **add the letter r** on Windows
- **Doubling backslash**

```
>>>
>>> myfile = open(r'C:\new\text.dat', 'w')
>>>
>>> myfile = open('C:\\new\\text.dat', 'w')
>>>
>>> path = r'C:\new\text.dat'
>>> path
'C:\\new\\text.dat'
>>> print(path)
C:\new\text.dat
>>>
```

1.4 Triple Quotes Code Multiline Block Strings

- Python also has a **triple-quoted string literal format**, sometimes called a **block string**

```
>>>
>>> mantra = """Always look
...  on the bright
... side of life."""
>>> mantra
'Always look\n on the bright\nside of life.'
>>> print(mantra)
Always look
 on the bright
side of life.
>>>
```

1.4 Triple Quotes Code Multiline Block Strings (con't)

- Triple-quoted strings are used for **documentation strings**, which are string literals that are taken **as comments.**
- triple-quoted strings are also sometimes used as a **"horribly hackish"** way to temporarily disable lines of code

```
>>>
>>> X = 1
>>> """
... import os
... print(os.getcwd())
... """
'\nimport os\nprint(os.getcwd())\n'
>>> Y = 2
>>>
```

2 Strings in Action

- Basic Operations
- Indexing and Slicing
- String Conversion Tools
- Changing Strings

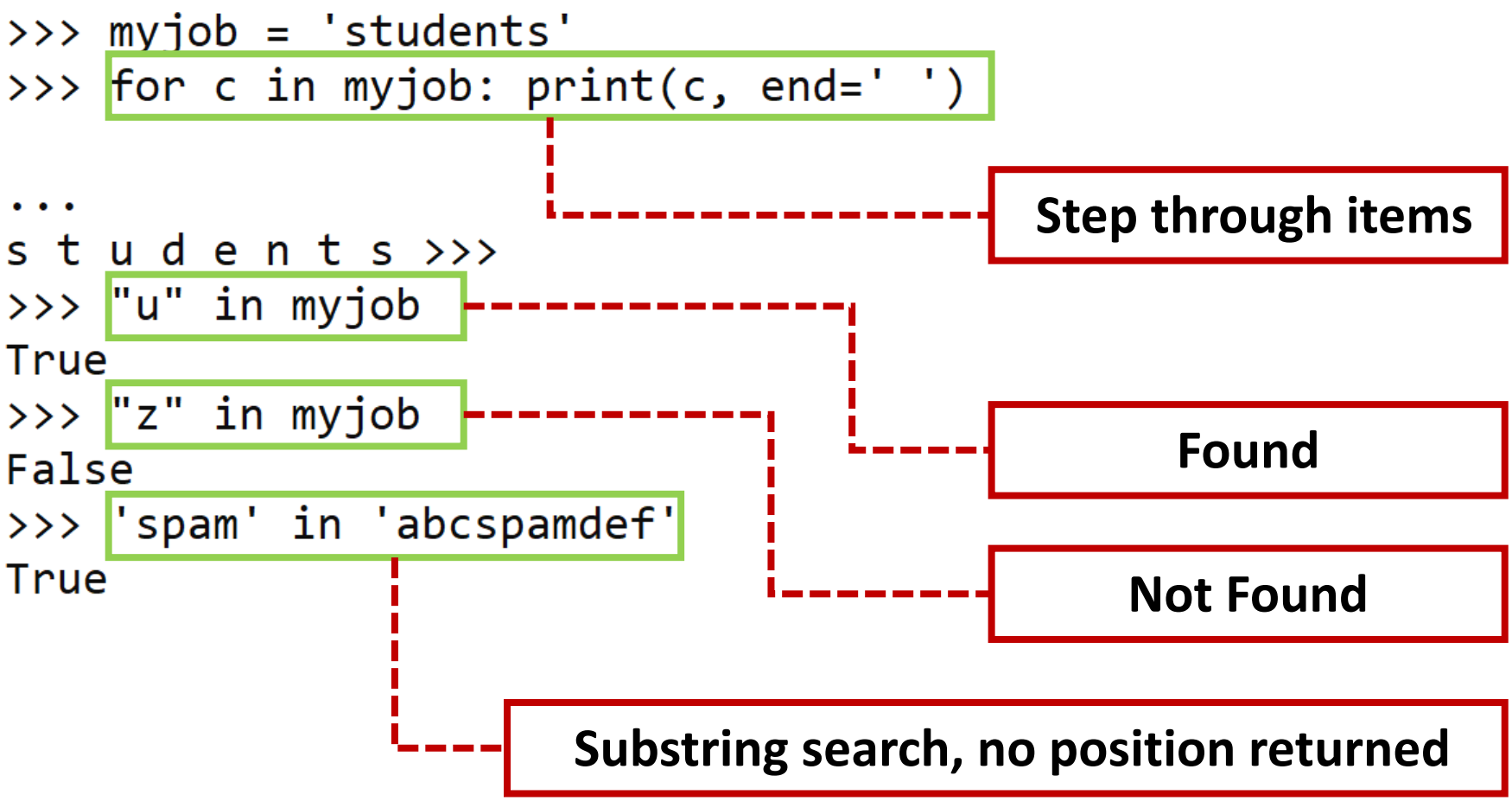
2.1 Basic Operations

- Strings can be concatenated using the + operator and repeated using the * operator:

```
>>> len('abc')
3
>>> 'abc' + 'def'
'abcdef'
>>> 'Py!' * 4
'Py!Py!Py!Py!'
>>> print('----------------------------------
----------------------------------------
-----')
----------------------------------------
----------------------------------------
>>> print('-' * 80)
----------------------------------------
----------------------------------------
```

2.1 Basic Operations (con't)

• . For substrings, in is much like the str.find() method



2.2 Indexing and Slicing

- Can access their components by position
- Characters in a string are **fetched by indexing**
  - providing the **numeric offset** of the desired component in **square brackets** after the string

  => get back the **one-character string** at the specified position

2.2 Indexing and Slicing

0	1	2	3
s	p	a	m
-4	-3	-2	-1

```
>>> S = 'spam'
>>>
>>> S[0], S[-2]
('s', 'a')
>>>
>>> S[1:3], S[1:], S[:-1]
('pa', 'pam', 'spa')
```

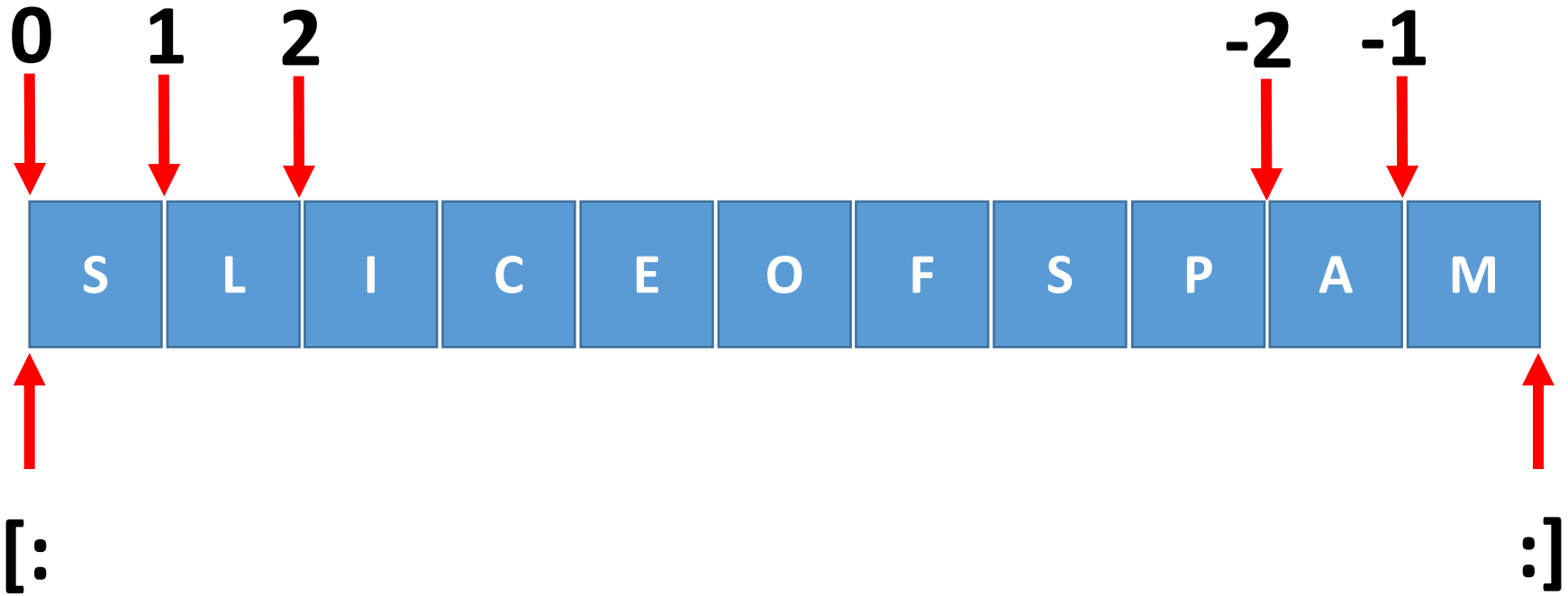
Indexing from front or end

Slicing: extract a section

Offsets and slices

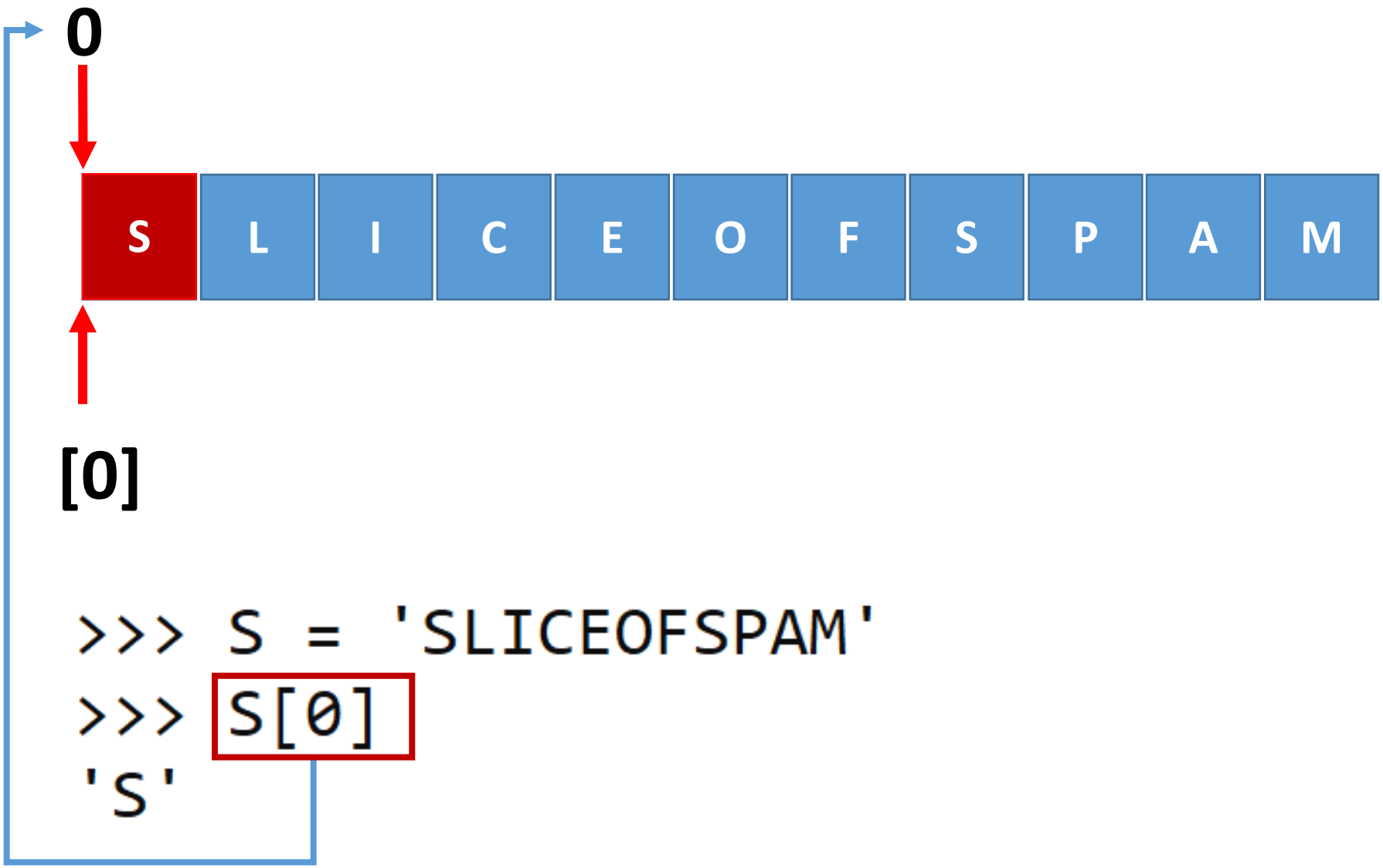
[start:end]

Indexes refer to places the knife “cuts.”



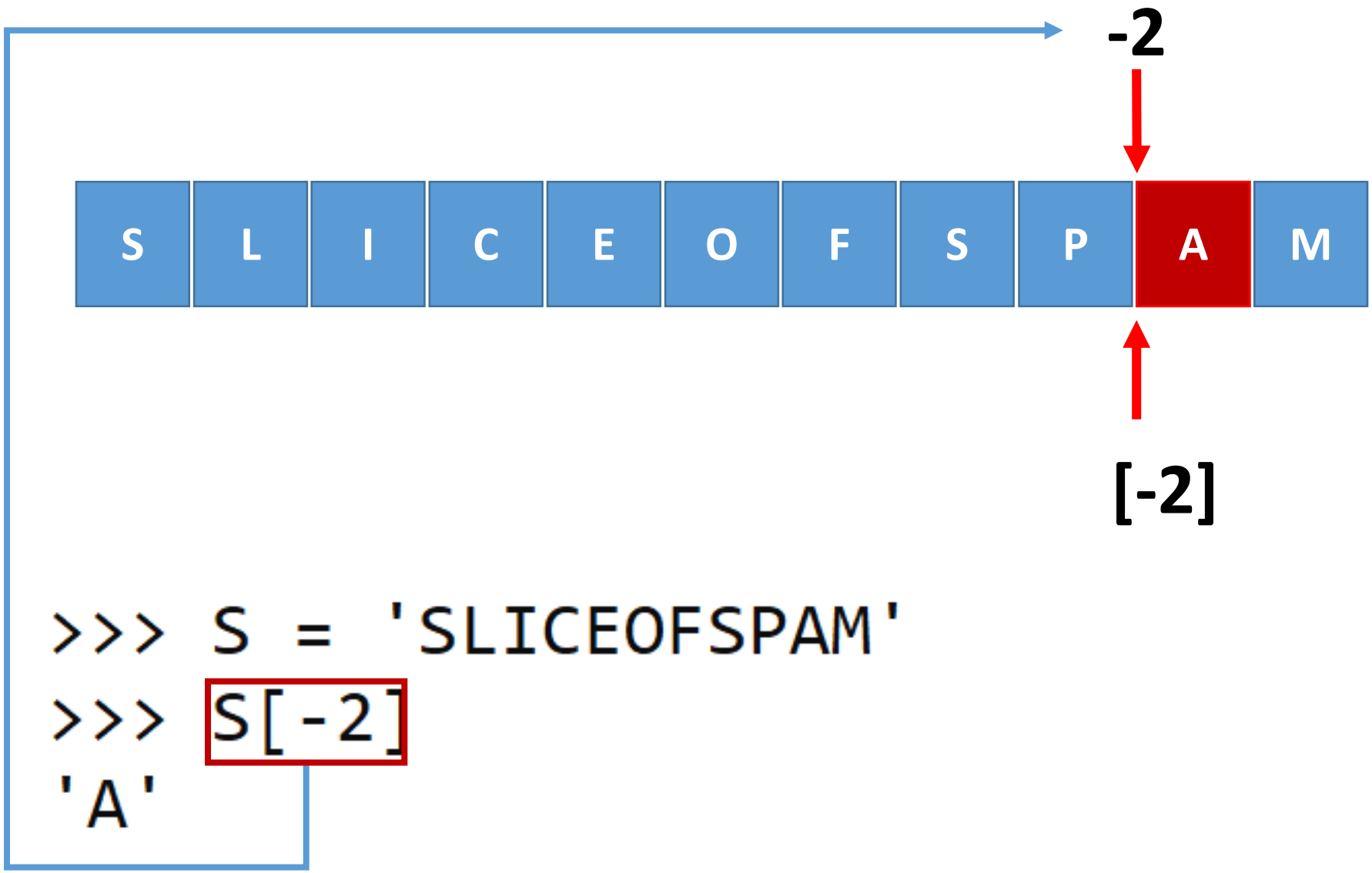
Default are beginning of sequences and end of sequences

Offsets and slices (con't)



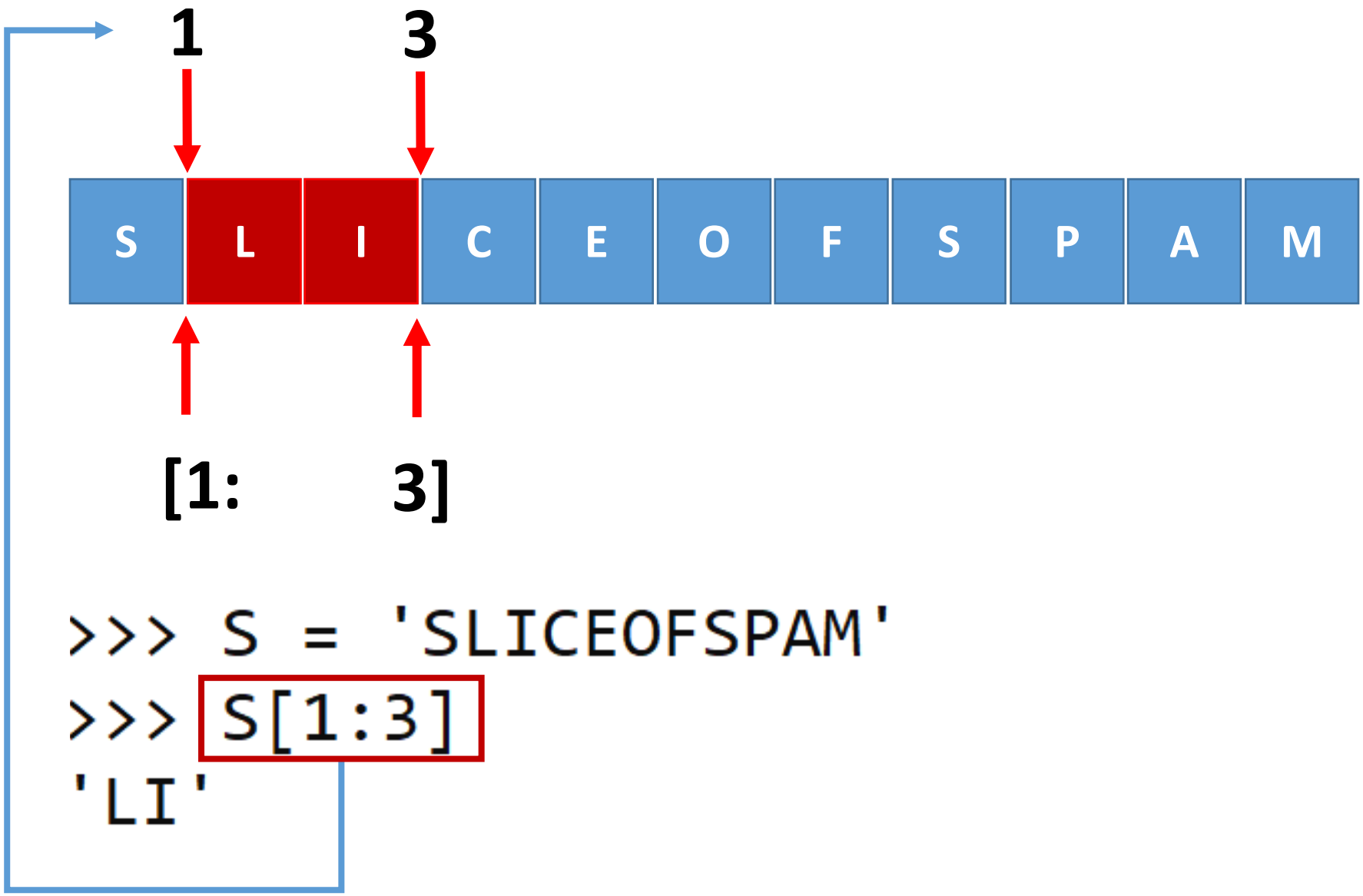

```
>>> S = 'SLICEOFSPAM'
>>> S[0]
'S'
```

Offsets and slices (con't)




```
>>> S = 'SLICEOFSPAM'
>>> S[-2]
'A'
```

Offsets and slices (con't)

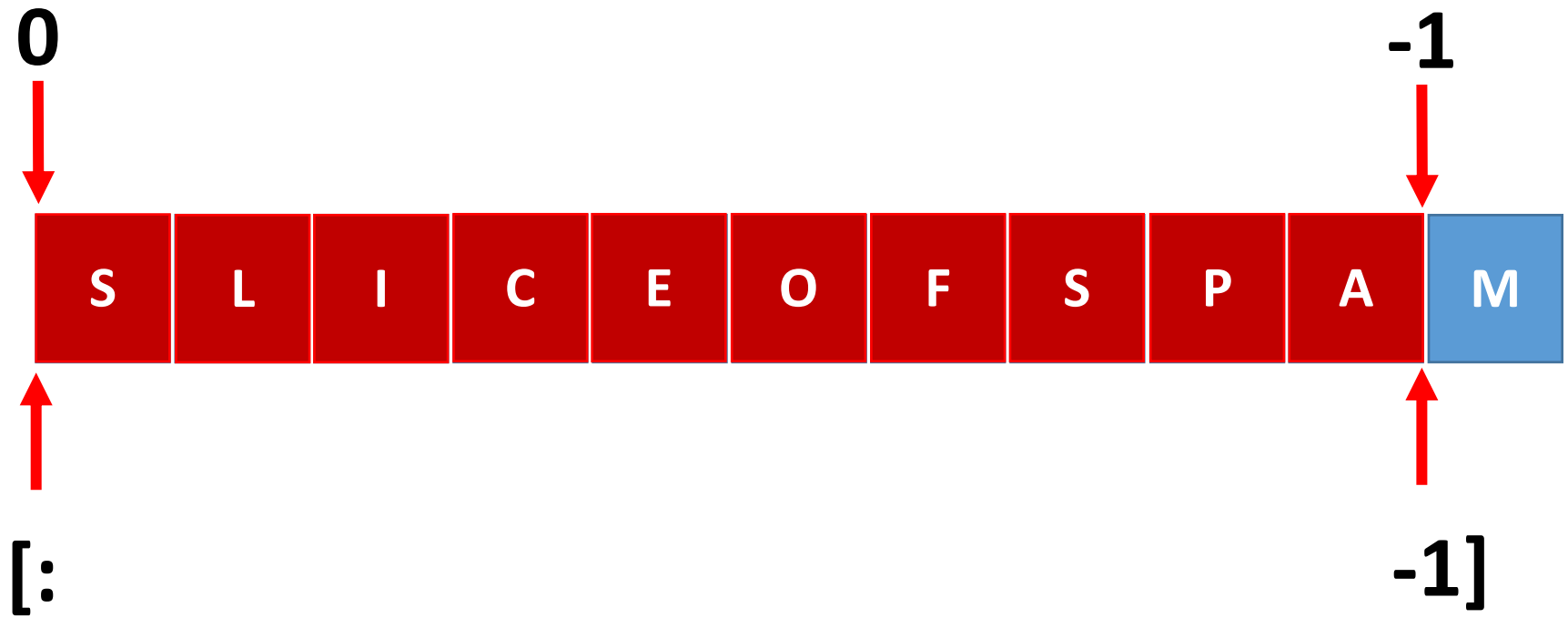



```
>>> S = 'SLICEOFSPAM'
>>> S[1:3]
'LI'
```

Offsets and slices (con't)


```
>>> S = 'SLICEOFSPAM'
>>> S[1:]
'LICEOFSPAM'
```

Offsets and slices (con't)




```
>>> S = 'SLICEOFSPAM'
>>> S[1:]
'LICEOFSPAM'
```

Summary of the details for reference

Indexing (S[*i*]) fetches components at offsets:

- The **first item** is at offset **0**.
- **Negative indexes** mean to count **backward** from the **end or right**.
- **S[0]** fetches the **first item**.
- **S[−2] fetches** the **second item** from the end (like S[len(S)−2]).

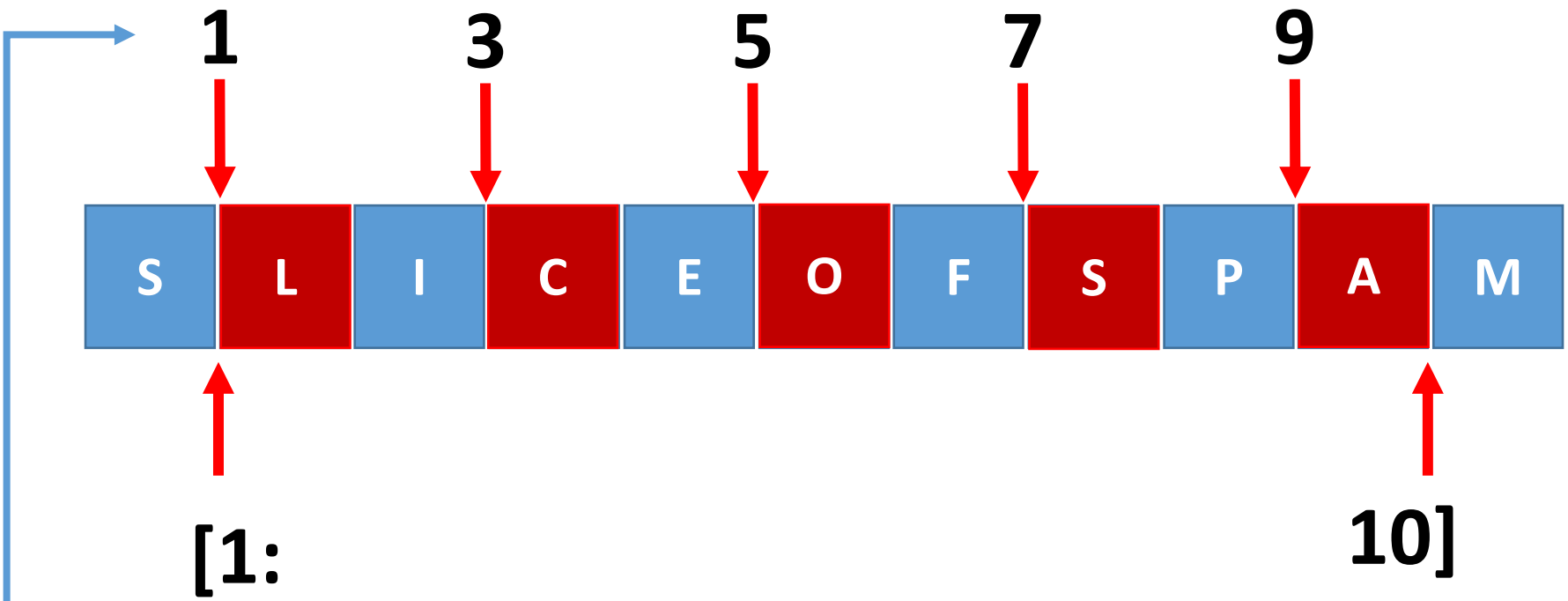
Summary of the details for reference (con't)

- Slicing (S[i:j]) extracts contiguous sections of sequences:
  - The **upper bound** is **noninclusive**. **"]"**
  - Slice boundaries default to 0 and the sequence length, if omitted.
  - S[1:3] fetches items at offsets 1 up to but not including 3.
  - S[1:] fetches items at offset 1 through the end (the sequence length).
  - S[:3] fetches items at offset 0 up to but not including 3.
  - S[:−1] fetches items at offset 0 up to but not including the last item.
  - S[:] fetches items at offsets 0 through the end—this effectively performs a toplevel copy of S.

Extended slicing: the third limit and slice objects

- Slice expressions support third index, used as a step.
- The full-blown form of a slice is now X[I:J:K]
  - extract all the items in X,
  - from offset I through J−1, by K.

Offsets and slices (con't)




```
>>> S = 'SLICEOFSPAM'
>>> S[1:10:2]
'LCOSA'
```

String Conversion Tools

- **can't mix strings** and **number** types around operators such as +
- **convert operands** before that operation if needed

```
>>> S = "42"
>>> I = 1
>>> S + I
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
TypeError: can only concatenate str (not "int") to str
>>> int(S) + I
43
>>> S + str(I)
'421'
```

String Conversion Tools (con't)

- Similar built-in functions handle floating-point number conversions to and from strings:

```
>>> str(3.1415), float("1.5")
('3.1415', 1.5)
>>> text = "1.234E-10"
>>> float(text)
1.234e-10
```

Character code conversions

- Possible to convert a **single character** to its underlying **ASCII integer code** by passing it to the built-in ***ord*** function.
- This returns the actual binary value of the corresponding byte in memory.
- The ***chr*** function performs the inverse operation, taking an **ASCII integer code** and converting it to the corresponding **character**

Character code conversions (con't)

Dec	Oct	Hex	C	Dec	Oct	Hex	C	Dec	Oct	Hex	C	Dec	Oct	Hex	C
0	0	0	^@	32	40	20		64	100	40	@	96	140	60	`
1	1	1	^A	33	41	21	!	65	101	41	A	97	141	61	a
2	2	2	^B	34	42	22	"	66	102	42	B	98	142	62	b
3	3	3	^C	35	43	23	#	67	103	43	C	99	143	63	c
4	4	4	^D	36	44	24	$	68	104	44	D	100	144	64	d
5	5	5	^E	37	45	25	%	69	105	45	E	101	145	65	e
6	6	6	^F	38	46	26	&	70	106	46	F	102	146	66	f
7	7	7	^G	39	47	27	'	71	107	47	G	103	147	67	g
8	10	8	^H	40	50	28	(	72	110	48	H	104	150	68	h
9	11	9	^I	41	51	29	)	73	111	49	I	105	151	69	i
10	12	a	^J	42	52	2a	*	74	112	4a	J	106	152	6a	j
11	13	b	^K	43	53	2b	+	75	113	4b	K	107	153	6b	k
12	14	c	^L	44	54	2c	,	76	114	4c	L	108	154	6c	l
13	15	d	^M	45	55	2d	-	77	115	4d	M	109	155	6d	m
14	16	e	^N	46	56	2e	.	78	116	4e	N	110	156	6e	n
15	17	f	^O	47	57	2f	/	79	117	4f	O	111	157	6f	o
16	20	10	^P	48	60	30	0	80	120	50	P	112	160	70	p
17	21	11	^Q	49	61	31	1	81	121	51	Q	113	161	71	q
18	22	12	^R	50	62	32	2	82	122	52	R	114	162	72	r
19	23	13	^S	51	63	33	3	83	123	53	S	115	163	73	s
20	24	14	^T	52	64	34	4	84	124	54	T	116	164	74	t
21	25	15	^U	53	65	35	5	85	125	55	U	117	165	75	u
22	26	16	^V	54	66	36	6	86	126	56	V	118	166	76	v
23	27	17	^W	55	67	37	7	87	127	57	W	119	167	77	w
24	30	18	^X	56	70	38	8	88	130	58	X	120	170	78	x
25	31	19	^Y	57	71	39	9	89	131	59	Y	121	171	79	y
26	32	1a	^Z	58	72	3a	:	90	132	5a	Z	122	172	7a	z
27	33	1b	^[	59	73	3b	;	91	133	5b	[	123	173	7b	{
28	34	1c	^\	60	74	3c	<	92	134	5c	\	124	174	7c	|
29	35	1d	^]	61	75	3d	=	93	135	5d	]	125	175	7d	}
30	36	1e	^^	62	76	3e	>	94	136	5e	^	126	176	7e	~
31	37	1f	^_	63	77	3f	?	95	137	5f	_	127	177	7f	

ASCII Table

Character code conversions (con't)

```
>>> ord('s')
115
>>> chr(115)
's'
>>> S = '5'
>>> S = chr(ord(S) + 1)
>>> S
'6'
>>> S = chr(ord(S) + 1)
>>> S
'7'
```

Character code conversions (con't)

- Convert binary digits to integer with ord

```
>>> B = '1101'
>>> I = 0
>>> while B!= '':
...     I = I * 2 + (ord(B[0]) - ord('0'))
...     B = B[1:]
...
>>> I
13
```

Changing Strings

- Remember the term “immutable sequence”? The immutable part means that you can’t change a string in-place

```
>>> S = 'spam'
>>> S[0] = "x"
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
TypeError: 'str' object does not support ite
m assignment
```

Changing Strings (con't)

- To change a string, you need to **build** and **assign** a **new string** using tools such as **concatenation** and **slicing**

```
>>> S = 'spam'
>>> S
'spam'
>>> S = S + 'SPAM!'
>>> S
'spamSPAM!'
>>> S = s[:4] + 'Burger' + S[-1]
>>> S
'a\nb\tBurger!'
>>> print(S)
a
b       Burger!
```

Changing Strings (con't)

- Can achieve similar effects with string method calls like ***replace***

```
>>> S = 'splot'
>>> S = S.replace('pl', 'pamal')
>>> S
'spamalot'
```

3 String Methods

- Methods are simply functions that are associated with particular objects.

Table 7-3. String method calls in Python 3.0

S.capitalize()	S.ljust(width [, fill])
S.center(width [, fill])	S.lower()
S.count(sub [, start [, end]])	S.lstrip([chars])
S.encode([encoding [,errors]])	S.maketrans(x[, y[, z]])
S.endswith(suffix [, start [, end]])	S.partition(sep)
S.expandtabs([tabsize])	S.replace(old, new [, count])
S.find(sub [, start [, end]])	S.rfind(sub [,start [,end]])
S.format(fmtstr, *args, **kwargs)	S.rindex(sub [, start [, end]])
S.index(sub [, start [, end]])	S.rjust(width [, fill])

3 String Methods (con't)

Table 7-3. String method calls in Python 3.0

S.isalnum()
S.isalpha()
S.isdecimal()
S.isdigit()
S.isidentifier()
S.islower()
S.isnumeric()
S.isprintable()
S.isspace()
S.istitle()
S.isupper()
S.join(iterable)
S.rpartition(sep)
S.rsplit([sep[, maxsplit]])
S.rstrip([chars])
S.split([sep [,maxsplit]])
S.splitlines([keepends])
S.startswith(prefix [, start [, end]])
S.strip([chars])
S.swapcase()
S.title()
S.translate(map)
S.upper()
S.zfill(width)

3.1 String Method Examples: Changing Strings

- **Construct a new string** with operations such as slicing and concatenation. Example: Replace '***mm***' in spammy by '**xx**'

```
>>> S = 'spammy'
>>> S = S[:3] + 'xx' + S[5:]
>>> S
'spaxxy'
```

```
>>> S = 'spammy'
>>> S = S.replace('mm', 'xx')
>>> S
'spaxxy'
```

3.1 String Method Examples: Changing Strings (con't)

Replace one fixed-size string that can occur at any offset, you can do a replacement again, or search for the substring with the string **find** method and then slice:

```
>>> S = 'xxxxSPAMxxxxSPAMxxxx'
>>> where = S.find('SPAM')
>>> where
4
>>> S = S[:where] + 'EGGS' + S[(where+4):]
>>> S
'xxxxEGGSxxxxSPAMxxxx'
```

The ***find*** method returns the ***offset*** where the substring appears or ***−1*** if it is not found.

3.2 String Method Examples: Parsing Text

Common role for string methods is as a simple form of text **parsing**, *analyzing structure and extracting substring*. To extract substrings at fixed offsets, we can employ slicing techniques:

```
>>> line = 'aaa bbb ccc'
>>> col1 = line[0:3]
>>> col2 = line[8:]
>>> col1
'aaa'
>>> col2
'ccc'
```

```
>>> line = 'aaa bbb ccc'
>>> cols = line.split()
>>> cols
['aaa', 'bbb', 'ccc']
```

3.2 String Method Examples: Parsing Text (con't)

- The string split method chops up a string into a list of substrings, around a **delimiter** string.

```
>>> line = 'bob,hacker,40'
>>> line.split(',')
['bob', 'hacker', '40']
>>>
>>> line = "i'mSPAMaSPAMlumberjack"
>>> line.split("SPAM")
["i'm", 'a', 'lumberjack']
```

3.4 Other Common String Methods in Action

rstrip(); upper(); isalpha(); endswith(); startswith()

```
>>> line ="The knights who say Ni!\n"
>>> line.rstrip()
'The knights who say Ni!'
>>> line.upper()
'THE KNIGHTS WHO SAY NI!\n'
>>> line.isalpha()
False
>>> line.endswith('Ni!\n')
True
>>> line.startswith('The')
True
```

4 String Formatting Expressions

- string formatting allows us to **perform multiple type-specific substitutions** on a string in a single step.
- The **% operator** provides a compact way to code multiple string substitutions all at once, instead of building and concatenating parts individually

```
>>> d1 = 1
>>> s1 = 'dead'
>>> 'That is ' + str(d1) + ' ' + s1 +' bird!'
'That is 1 dead bird!'
```

4 String Formatting Expressions (con't)

- To format strings:

  1. On the **left** of the **%** operator, provide a **format string** containing one or more embedded conversion targets**, each of which starts** with a **%** (e.g., %d).

  2. On the **right** of the **%** operator, provide the **object** that you want Python to insert into the format string on the left in place of the conversion target.

```
>>> d1 = 1
>>> s1 = 'dead'
>>> 'That is %d %s bird!' % (d1, s1)
'That is 1 dead bird!'
```

4 String Formatting Expressions (con't)

Table 7-4. String formatting type codes

Code	Meaning
s	String (or any object's str(X) string)
r	s, but uses repr, not str
c	Character
d	Decimal (integer)
i	Integer
u	Same as d (obsolete: no longer unsigned)
o	Octal integer
x	Hex integer
X	x, but prints uppercase

4 String Formatting Expressions (con't)

Table 7-4. String formatting type codes

Code	Meaning
e	Floating-point exponent, lowercase
E	Same as e, but prints uppercase
f	Floating-point decimal
F	Floating-point decimal
g	Floating-point e or f
G	Floating-point E or F
%	Literal %

Test Your Knowledge: Quiz

1. Can the string find method be used to search a list?
2. Can a string slice expression be used on a list?
3. How would you convert a character to its ASCII integer code? How would you convert the other way, from an integer to a character?
4. How might you go about changing a string in Python?
5. Given a string S with the value "s,pa,m", name two ways to extract the two characters in the middle.
6. How many characters are there in the string "a\nb\x1f\000d"?
7. Why might you use the string module instead of string method calls?