Strings

CSC 1200 - Principles of Computing

Overview

- Accessing Characters
- The len Function
- String Slices
  - Special Slices
- Searching a String
- String Traversal
- String Methods
- The in Operator
- Relational Operators

Accessing Characters in a String

- A **string** is a sequence of characters. For example, 'CSC1200' is a sequence of 7 characters.
- To access individual characters in a string, we use the bracket operator
  - course = 'CSC1200'
  - The 7 characters of the string are indexed 0 - 6
  - course[1] is 'S'
  - course[4] is '2'
  - course[7] will give an error (Index out of Range)
  - course[-1] will NOT give an error. The string wraps around backward, so course[-1] is '0'
  - course[-4] is '1'
  - The value used as an index MUST be an integer. (course[2.5] will give a Type Error)

The len Function

- The built-in function len returns the number of characters in a string.

```
>>> name = 'Ricardo'
>>> length = len(name)
>>> length
7
>>> first = name[0]
>>> first
'R'
>>> last = name[len(name)]
Traceback (most recent call last):
  File "<pyshell#11>", line 1, in <module>
    last = name[len(name)]
IndexError: string index out of range
>>> last = name[len(name)-1]
>>> last
'o'
>>>
```

String Slices

- A segment of a string is called a **slice**.
- We use the bracket operator, similar to selecting a single character, except a range of indices is given using a colon operator.
- string[n:m] returns the part of the string starting with the nth character up to (but not including) the mth character.

```
>>> mascot = "golden eagle"
>>> mascot[0:6]
'golden'
>>> mascot[6:5]
''
>>> mascot[6:11]
' eagl'
>>> mascot[7:12]
'eagle'
>>> mascot[12]
Traceback (most recent call last):
  File "<pyshell#19>", line 1, in <module>
    mascot[12]
IndexError: string index out of range
>>>
```

Special Slices

- If you omit the first index, the slice starts at the beginning of the string.

```
>>> mascot[:8]
'golden e'
```

- If you omit the last index, the slice continues to the end of the string.

```
>>> mascot[3:]
'den eagle'
```

- If the first index is greater than or equal to the second, the result is an **empty string**.

```
>>> mascot[5:5]
''
>>> mascot[5:1]
''
>>> mascot[5:6]
'n'
```

Strings are Immutable

- Strings are immutable, which means they can't be changed.
- Suppose we have a string, word = 'quiat' and we realized that we really want character 3 to be 'e' (so that the word is 'quiet').
  - It's tempting to think that we can use word[3] = 'e' to change the spelling

```
>>> word = 'quiat'
>>> word[3] = 'e'
Traceback (most recent call last):
  File "<pyshell#26>", line 1, in <module>
    word[3] = 'e'
TypeError: 'str' object does not support item assignment
```

- We would instead have to construct a new string and assign the new string to word.

```
>>> word = word[:3]+'e'+word[4:]
>>> word
'quiet'
```

Searching a String

- Searching a sequence for a particular item is a common thing to do.
- For example, we might want to see if a string contains a particular letter or sequence of letters.
- Two built-in functions that come in handy when dealing with characters are ord and chr
  - *ord('c')* returns the ASCII value of the character c
  - *chr(a)* returns the character that has ASCII value a

String Traversal

- We often need to process a string character by character. Moving sequentially through the string one character at a time is called a **traversal**.

Example: print the characters of a string backwards

While loop version:

```
def print_backwards( string ):
    curr_index = len(string) - 1
    while curr_index >= 0:
        print(string[curr_index],end='')
        curr_index -= 1
    print()


print_backwards('backwards')
print_backwards('yo banana boy')
```

Output:

```
sdrawkcab
yob ananab oy
>>>
```

For loop version:

```
def print_backwards( string ):
    for curr_index in range(len(string)-1, -1, -1):
        print(string[curr_index],end='')

    print()

print_backwards('backwards')
print_backwards('yo banana boy')
```

String Methods

- A method is a function that belongs to a particular type or object, and we must use dot notation to access the method.
- Some useful string methods include:
  - upper - returns the string converted to all uppercase
  - lower - returns the string converted to all lowercase
  - isupper - returns True if all cased characters are uppercase
  - islower - returns True if all cased characters are lowercase
  - isalpha - returns True if all characters are alphabetic
  - isdigit - returns True if all characters are numeric
  - find - returns the lowest index where a substring is found (returns -1 if substring is not found)
  - index - like find, except that a ValueError is raised if the substring is not found

Examples

```
>>> word = 'QUit'
>>> uc = word.upper()
>>> uc
'QUIT'
>>> lc = word.lower()
>>> lc
'quit'
>>> word.isupper()
False
>>> uc.isupper()
True
>>> word.islower()
False
>>> lc.islower()
True
```

```
>>> pwd = 'password123'
>>> number = '123'
>>> name = 'Amelia'
>>> pwd.isalpha()
False
>>> pwd.isdigit()
False
>>> number.isalpha()
False
>>> number.isdigit()
True
>>> name.isalpha()
True
>>> name.isdigit()
False
>>> pwd.isalnum()
True
```

Examples (Continued)

```
>>> word = 'vegetable'
>>> word.find('table')
4
>>> word.find('get')
2
>>> word.find('e')
1
>>> word.find('egg')
-1
>>> word.index('able')
5
>>> word.index('tale')
Traceback (most recent call last):
  File "<pyshell#54>", line 1, in <modu
    word.index('tale')
ValueError: substring not found
```

The in Operator

- The word in is a Boolean operator. For strings, it returns True if the first string is a substring of the second.
  - 'a' in 'banana' returns True
  - 'seed' in 'banana' returns False

```
>>> def in_both( word1, word2 ):
...     for letter in word1:
...         if letter in word2:
...             print(letter)
...
...
>>> in_both('banana', 'apple')
a
a
a
>>> in_both('apples','oranges')
a
e
s
```

```
>>> in_both('apple', 'banana')
a
```

Relational Operators for Strings

- Several relational operators can be used with strings.
  - To check for equality: ==
  - To check if a string comes alphabetically before another: <
  - To check if a string comes alphabetically after another: >
- Note that ordering of characters is based on ASCII

```
>>> fruit = 'apple'
>>> fruit == 'apple'
True
>>> fruit != 'banana'
True
>>> fruit < 'banana'
True
>>> fruit < 'Banana'
False
```

```
>>> fruit > 'aardvark'
True
>>> fruit > 'App'
True
```

Hex	Value	Hex	Value	Hex	Value	Hex	Value	Hex	Value	Hex	Value	Hex	Value	Hex	Value
00	NUL	10	DLE	20	SP	30	0	40	@	50	P	60	`	70	p
01	SOH	11	DC1	21	!	31	1	41	A	51	Q	61	a	71	q
02	STX	12	DC2	22	"	32	2	42	B	52	R	62	b	72	r
03	ETX	13	DC3	23	#	33	3	43	C	53	S	63	c	73	s
04	EOT	14	DC4	24	$	34	4	44	D	54	T	64	d	74	t
05	ENQ	15	NAK	25	%	35	5	45	E	55	U	65	e	75	u
06	ACK	16	SYN	26	&	36	6	46	F	56	V	66	f	76	v
07	BEL	17	ETB	27	'	37	7	47	G	57	W	67	g	77	w
08	BS	18	CAN	28	(	38	8	48	H	58	X	68	h	78	x
09	HT	19	EM	29	)	39	9	49	I	59	Y	69	i	79	y
0A	LF	1A	SUB	2A	*	3A	:	4A	J	5A	Z	6A	j	7A	z
0B	VT	1B	ESC	2B	+	3B	;	4B	K	5B	[	6B	k	7B	{
0C	FF	1C	FS	2C	,	3C	<	4C	L	5C	\	6C	l	7C	|
0D	CR	1D	GS	2D	-	3D	=	4D	M	5D	]	6D	m	7D	}
0E	SO	1E	RS	2E	.	3E	>	4E	N	5E	^	6E	n	7E	~
0F	SI	1F	US	2F	/	3F	?	4F	O	5F	_	6F	o	7F	DEL