

Special Topics in Computer Science- CSC 4992

Sequences: 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

Examples

```
a = 'apple'
b = 'banana'
print(a, b)                                # Displays apple banana

fruits = (a, b)                             # A tuple
print(fruits)                             # Displays ('apple', 'banana')

veggies = ['bean', 'lettuce']              # A list
print(veggies)                             # Displays ['bean', 'lettuce']
```

Strings contains characters

Tuples and lists can contain anything

The **len** Function

'Hi there!'

H	i		t	h	e	r	e	!
0	1	2	3	4	5	6	7	8

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

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

```
9
```

```
>>> s = 'Hi there!'
```

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

```
 '!'
```

Positions or Indexes

'Hi there!'

H	i		t	h	e	r	e	!
0	1	2	3	4	5	6	7	8

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!'

H	i		t	h	e	r	e	!
0	1	2	3	4	5	6	7	8

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!
```

Summing with Strings

'Hi there!'

H	i		t	h	e	r	e	!
0	1	2	3	4	5	6	7	8

Start with an empty string and add characters to it with +

```
noVowels = ''
for ch in 'Hi there!':
    if not ch in ('a', 'e', 'i', 'o', 'u',
                  'A', 'E', 'I', 'O', 'U'):
        noVowels += ch
print(noVowels)

# Prints H thr!
```

The Subscript Operator

'Hi there!'

H	i		t	h	e	r	e	!
0	1	2	3	4	5	6	7	8

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 Subscript Operator

- The len function tells us how many characters are in a string.
- Notice the relationship between the length of a string and the index of the last character in the string.

```
>>> name = "PYTHON ROCKS"
>>> name
'PYTHON ROCKS'
>>> name[0]
'P'
>>> first = name[0]
>>> first
'P'
>>> length = len("PYTHON ROCKS")
>>> length
12
>>> length2 = len(name)
>>> length2
12
>>>
```

The Subscript Operator

- We can go one step further and combine the range function with the len function to allow us to access each character in a string.

```
>>> name = "PYTHON ROCKS"  
>>> length = len(name)  
>>> for i in range (length):  
        print (name[i])
```

```
P  
Y  
T  
H  
O  
N  
  
R  
O  
C  
K  
S
```

```
>>>
```

An Index-Based Loop

'Hi there!'

H	i		t	h	e	r	e	!
0	1	2	3	4	5	6	7	8

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])
```

An Index-Based Loop

- Gets a particular characters in a string.

Positive indexes	0	1	2	3	4	5	6	7	8	9	10	11
String	P	Y	T	H	O	N		R	O	C	K	S
Negative indexes	-12	-11	-10	-9	-8	-7	-6	-5	-4	-3	-2	-1

```
>>> 'PYTHON ROCKS'[0]
'P'
>>> 'PYTHON ROCKS'[1]
'Y'
>>> 'PYTHON ROCKS'[11]
'S'
>>> 'PYTHON ROCKS'[-1]
'S'
>>> 'PYTHON ROCKS'[-12]
'P'
>>> 'PYTHON ROCKS'[12]
Traceback (most recent call last):
  File "<pyshell#8>", line 1, in <module>
    'PYTHON ROCKS'[12]
IndexError: string index out of range
```

Oddball Indexes

'Hi there!'

H	i		t	h	e	r	e	!
0	1	2	3	4	5	6	7	8

To get to the last character in a string:

```
s = 'Hi there!'
print(s[len(s) - 1])    # Displays !
```

Oddball Indexes

'Hi there!'

H	i		t	h	e	r	e	!
0	1	2	3	4	5	6	7	8

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

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*

non-printing characters

If inserted directly, are preceded by a backslash (the \ character)

- new line ' \n '
- tab ' \t '

Concatenation Operator (+)

- When + is applied to two strings, we call it the concatenation operator.

```
>>> "hello " + "world!"  
'hello world!'  
>>> fname = 'John'  
>>> lname = 'Smith'  
>>> fname + lname  
'JohnSmith'  
>>> fullName = fname + ' ' + lname  
>>> fullName  
'John Smith'  
>>>
```

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

Repetition Operator (*)

- Takes a string and repeats it as many times as you would like.
- Repetition operator has a higher precedence than concatenation.

```
>>> 'go'*3
'gogogo'
>>> 'go '*3 + 'twins '*2
'go go go twins twins '
>>> ('hello ' + 'world ')*3
'hello world hello world hello world '
>>>
```

Slicing Operator ([:])

- Similar to the index operator except that it can get multicharacter parts of the string.
- We often call these parts of a string substrings.
- The first number after the left square bracket gives the starting index of the first character in the substring.
- The number after the ':' is the index that is one after the last character selected for the substring.
- This is similar to the bounds of the range function where the upper bound is not included.

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

Slicing Operator ([:])

0	1	2	3	4	5	6	7	8	9	10	11
P	Y	T	H	O	N		R	O	C	K	S

```
'PYTHON ROCKS '  
>>> name[2:6]  
'THON '  
>>> name[10:12]  
'KS '  
>>> name[:6]  
'PYTHON '  
>>> name[7:]  
'ROCKS '  
>>> name[:]  
'PYTHON ROCKS '  
>>> |
```

```
>>> name
'PYTHON ROCKS'
>>> len(name)
12
>>> for i in range(len(name)):
        print(name[0:i])

P
PY
PYT
PYTH
PYTHO
PYTHON
PYTHON
PYTHON R
PYTHON RO
PYTHON ROC
PYTHON ROCK
```



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

Characters

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

Index

↑
First

↑
Last

Extended Slicing

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


my_str = 'hello world'

my_str[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



The diagram illustrates the slicing operation `helloString[::2]`. It shows a table with two rows: 'Characters' and 'Index'. The characters are 'H', 'e', 'l', 'l', 'o', ' ', 'W', 'o', 'r', 'l', 'd' and the indices are 0 through 10. Every second character (at indices 0, 2, 4, 6, 8, 10) is highlighted in light blue. Below the table, five curved arrows originate from the indices 0, 2, 4, 6, and 8, all pointing towards the index 2 cell, visually representing the step size of 2 in the slicing operation.

- how to make a copy of a string:

```
my_str = 'hi mom'  
new_str = my_str[:]
```

- how to reverse a string

```
my_str = "madam I'm adam"  
reverseStr = my_str[::-1]
```

Strings are Immutable

- strings are immutable, that is you cannot change one once you make it:
 - `a_str = 'spam'`
 - `a_str[1] = 'l' → ERROR`
- However, you can use it to make another string (copy it, slice it, etc.)
 - `new_str = a_str[:1] + 'l' + a_str[2:]`
 - `a_str → 'spam'`
 - `new_str → 'slam'`

Membership Operators

'in' and *'not in'*

- The *in* operator determines whether one string contains another.
- The *not in* operator determines whether one string does not contain another.

```
>>> name = 'PYTHON ROCKS'
>>> 'PY' in name
True
>>> 'z' in name
False
>>> 'THON' in name
True
>>> 'j' in 'john'
True
>>> 'i' not in 'john'
True
>>> 'j' not in 'john'
False
>>> 'jh' in 'john'
False
>>> first = 'j'
>>> first in 'john'
True
```

Operator	Example	Result	Description
+	"hello " + "world"	"hello world"	Concatenation , joins two strings together
*	"abc"*3	"abcabcabc"	Repetition , repeats the string
[i]	"abc"[1]	'b'	Index , returns the character at position i
[i:j]	"abcdef"[0:2]	"ab"	Slice returns the substring from i to j-1
len()	len('abc')	3	Determines the length of a string.
in	'bc' in 'abc'	True	Determines whether one string contains another
not in	'bc' not in 'abc'	False	Determines whether one string does not contain another

Enumerate Function

- The enumerate function prints out two values: the **index of an element** and **the element** itself
- Can use it to iterate through both the index and element simultaneously, doing dual assignment


```
>>> river = "Mississippi"
>>> for index, letter in enumerate (river):
    print(index, letter)
```

```
0 M
1 i
2 s
3 s
4 i
5 s
6 s
7 i
8 p
9 p
10 i
```

String Methods

Method	Use	Explanation
<code>center</code>	<code>astr.string.center(w)</code>	returns the string <code>astr.string</code> surrounded by spaces to make <code>astr.string</code> <code>w</code> characters long.
<code>count</code>	<code>astr.string.count(item)</code>	Returns the number of occurrences of <code>item</code> in <code>astr.string</code> .
<code>ljust</code>	<code>astr.string.ljust(w)</code>	Returns <code>astr.string</code> left justified in a field of width <code>w</code> .
<code>rjust</code>	<code>astr.string.rjust(w)</code>	Returns <code>astr.string</code> right justified in a field of width <code>w</code> .
<code>upper</code>	<code>astr.string.upper()</code>	Returns <code>astr.string</code> in all uppercase.
<code>lower</code>	<code>astr.string.lower()</code>	Returns <code>astr.string</code> in all lowercase.
<code>index</code>	<code>astr.string.index(item)</code>	Returns the index of the first occurrence of <code>item</code> in <code>astr.string</code> , or an error if not found.
<code>find</code>	<code>astr.string.find(item)</code>	Returns the index of the first occurrence of <code>item</code> in <code>astr.string</code> , or <code>-1</code> if not found.
<code>replace</code>	<code>astr.string.replace(old,new)</code>	Replaces all occurrences of <code>old</code> substring with <code>new</code> substring in <code>astr.string</code>

```
astring = "golden gopher football"
```

0	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15	16	17	18	19	20	21
g	o	l	d	e	n		g	o	p	h	e	r		f	o	o	t	b	a	l	l

```
>>> "hello".ljust(10)
'hello      '
>>> "hello" .rjust(10)
'      hello'
>>> "hello". center (10)
'  hello    '
>>> "hello". center (11)
'   hello   '
>>> "hello".ljust(3)
'hello'
>>> astring = "golden gopher football"
>>> astring.count('o')
4
>>> astring.count('oo')
1
>>> astring.find('b')
18
```

```
>>> astring.index('b')
18
>>> astring.find('oo')
15
>>> astring.find('badger')
-1
>>> astring.index('cyclone')
Traceback (most recent call last):
  File "<pyshell#71>", line 1, in <module>
    astring.index('cyclone')
ValueError: substring not found
>>> 'lab cd ef'.replace('cd' , 'xy')
'lab xy ef'
```

```
>>> 'john'.upper()
'JOHN'
>>> 'JAMES'.lower()
'james'
```

Chaining Methods

Methods can be chained together.

- Perform first operation, yielding an object
- Use the yielded object for the next method

```
my_str = 'Python Rules!'
```

```
my_str.upper() ⇒ 'PYTHON RULES!'
```

```
my_str.upper().find('O')
```

```
⇒ 4
```

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

```
a_str = 'He had the bat'
```

```
a_str.find('t') ⇒ 7 # 1st 't', start at 0
```

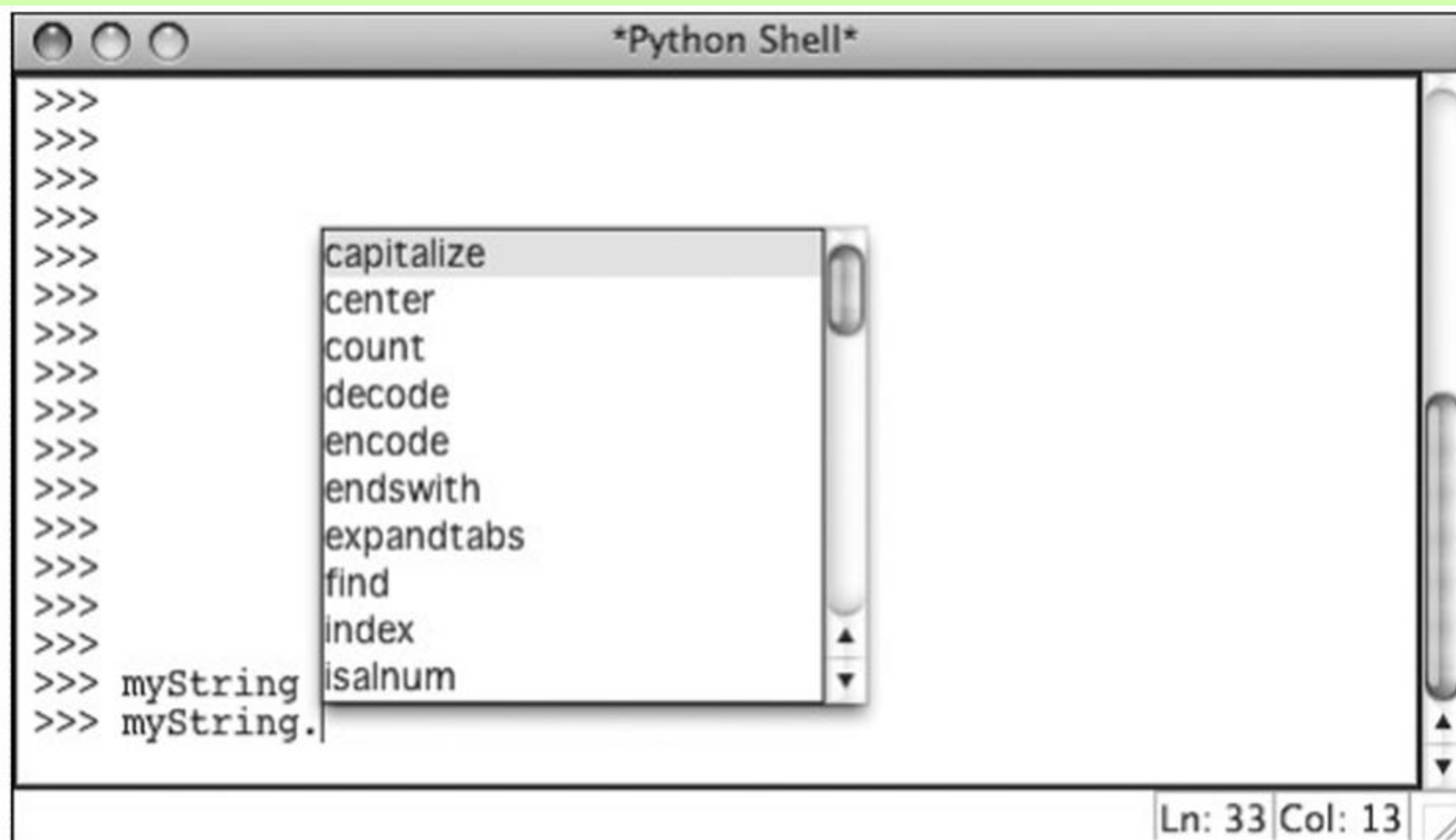
```
a_str.find('t',8) ⇒ 13 # 2nd 't'
```

Nesting Methods

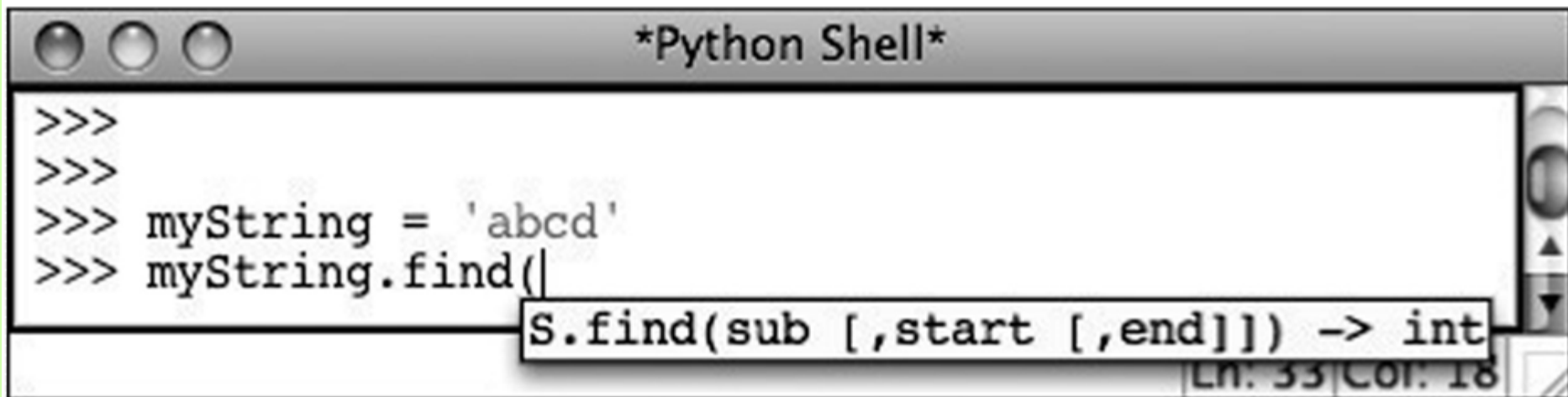
- You can “nest” methods, that is the result of one method as an argument to another
- remember that parenthetical expressions are did “inside out”: do the inner parenthetical expression first, then the next, using the result as an argument

`a_str.find('t', a_str.find('t')+1)`

- translation: find the second 't'.



In IDLE, tab lists potential methods.



The image shows a screenshot of a Python Shell window titled "*Python Shell*". The window contains a text area with the following code:

```
>>>  
>>>  
>>> myString = 'abcd'  
>>> myString.find(|
```

A pop-up window is displayed over the code, showing the signature of the `find` method:

```
S.find(sub [,start [,end]]) -> int
```

At the bottom right of the pop-up, the text "Ln: 33 Col: 18" is visible.

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

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 ASCII Character Set

ASCII Value	Char	ASCII Value	Char	ASCII Value	Char	ASCII Value	Char
32	' '	61	=	81	Q	105	i
33	!	62	>	82	R	106	j
34	"	65	A	83	S	107	k
42	*	66	B	84	T	108	l
43	+	67	C	85	U	109	m
45	-	68	D	86	V	110	n
47	/	69	E	87	W	111	o
48	0	70	F	88	X	112	p
49	1	71	G	89	Y	113	q
50	2	72	H	90	Z	114	r
51	3	73	I	97	a	115	s
52	4	74	J	98	b	116	t
53	5	75	K	99	c	117	u
54	6	76	L	100	d	118	v
55	7	77	M	101	e	119	w
56	8	78	N	102	f	120	x
57	9	79	O	103	g	121	y
60	<	80	P	104	h	122	z

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

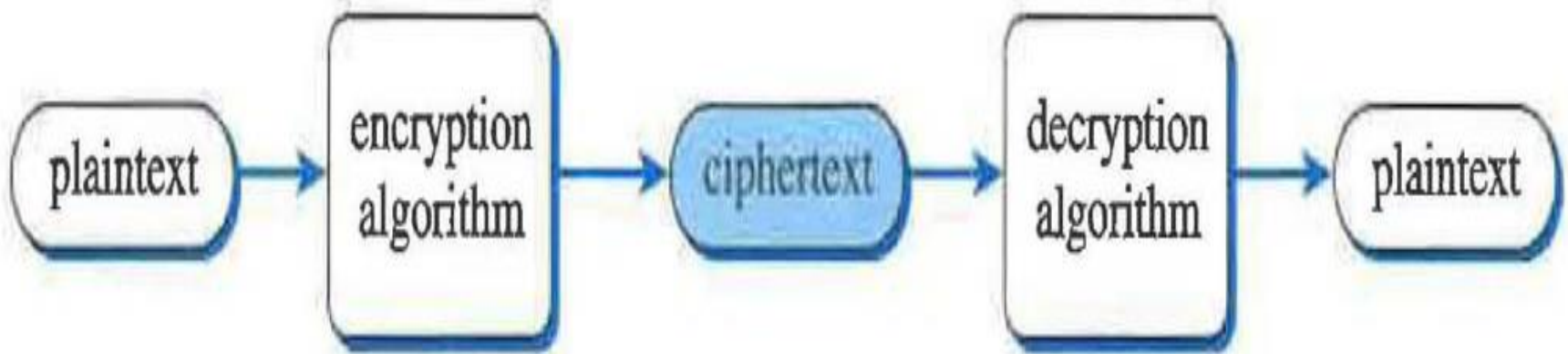
print(chr(65))            # Displays A

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

Encoding and Decoding Messages

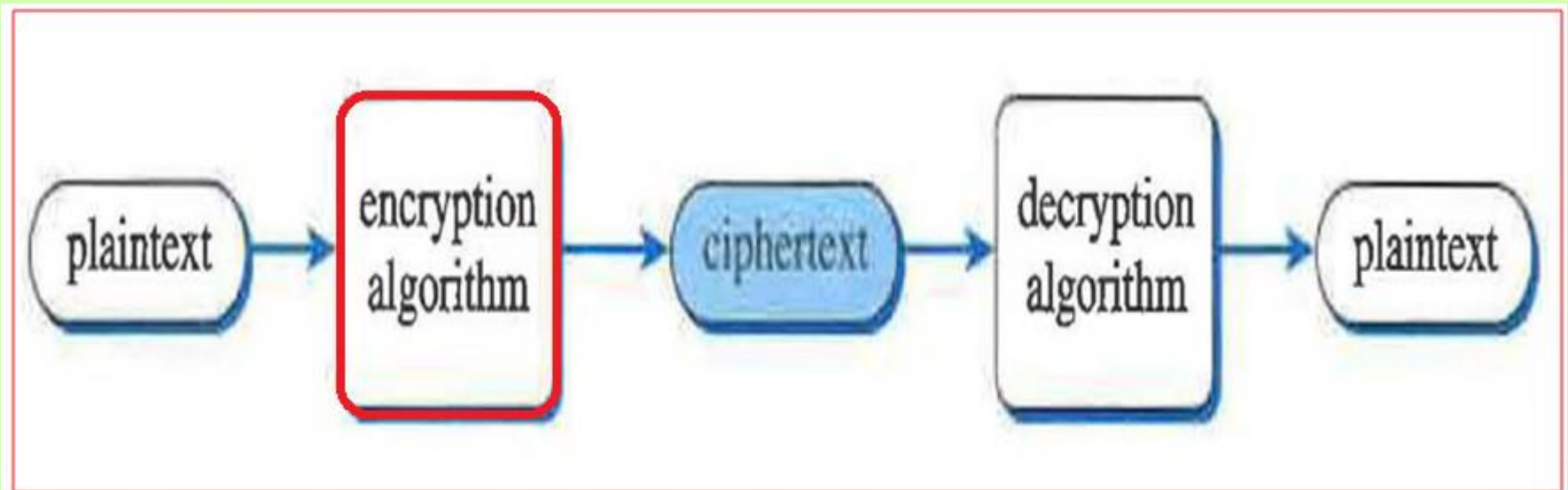
- Cryptography is the science of making messages secure, of transforming readable messages into unreadable messages and back again.
- Messages that are readable are called **plaintext**.
- Messages that are unreadable are called **ciphertext**.
- The process of turning plaintext into ciphertext is called **encryption**.
- The reverse process of turning ciphertext into plaintext is called **decryption**.

Encoding and Decoding Messages (cont'd.)



Encrypting Messages

- Write a Python function that takes the plaintext message as a parameter and returns the ciphertext message.



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 Friday!"
```

```
code = ""
```

```
for ch in source:
```

```
    code = code + str(ord(ch)) + " "
```

```
print(code)
```

```
# Displays 73 32 119 111 110 39 116 32 98 101 32 104 101 114 101 32 70  
# 114 105 100 97 121 33
```

Data Decryption

To decrypt an encoded message, we split it into a list of substrings and convert these ASCII values to the original characters

```
source = ""
for ascii in code.split():
    source = source + chr(int(ascii))
print(source)          # Displays I won't be here Friday!
```


Encrypting Messages (cont'd.)

- One of the easiest ways to encrypt a message is to simply scramble the letters.
- For example the word "apple" could be randomly transformed to "lapep."
- There are 120 different possible arrangements of the word "apple."
- Encryption and decryption algorithms must work together in some agreed upon way, with the encryption algorithm scrambling letters and the decryption algorithm unscrambling them.

Encrypting Messages (cont'd.)

- One way to scramble the letters of a message is to **separate the message** into two groups of characters.
- The first group composed of the **even-numbered** characters and the second group of the **odd-numbered** characters.
- If we create one string out of the even-numbered characters and another out of the odd, we can concatenate the two new strings together to form the **ciphertext** string.
- Because this results in a string with the characters shuffled to new positions, we call this a **transposition cipher**(rail fence cipher). (en.wikipedia.org/wiki/Transposition_cipher)

Transposition Cipher

Original	It was a dark and stormy night
Even	I _ a _ _ a k a d s o m _ i h
Odd	t w s a d r _ n _ t r y n g t

Break up the plaintext into even and odd characters

twsadr_n_tryngt + I_a__akadsom_ih

Combine the even and odd parts to make the ciphertext

Encrypting Using Transposition

```
def scramble2Encrypt(plainText):
    evenChars = ""
    oddChars = ""
    charCount = 0
    for ch in plainText:
        if charCount % 2 == 0:
            evenChars = evenChars + ch
        else:
            oddChars = oddChars + ch
        charCount = charCount + 1
    cipherText = oddChars + evenChars
    return cipherText
```

```
>>> scramble2Encrypt('abababab')
'bbbbaaaa'
>>> scramble2Encrypt('ababababc')
'bbbbaaaac'
>>> scramble2Encrypt('I do not like green eggs and ham')
' ontlk re gsadhmId o iegeneg n a'
>>> scramble2Encrypt('a')
'a'
>>> scramble2Encrypt(' ')
' '
```

Encrypting Using Transposition (cont'd.)

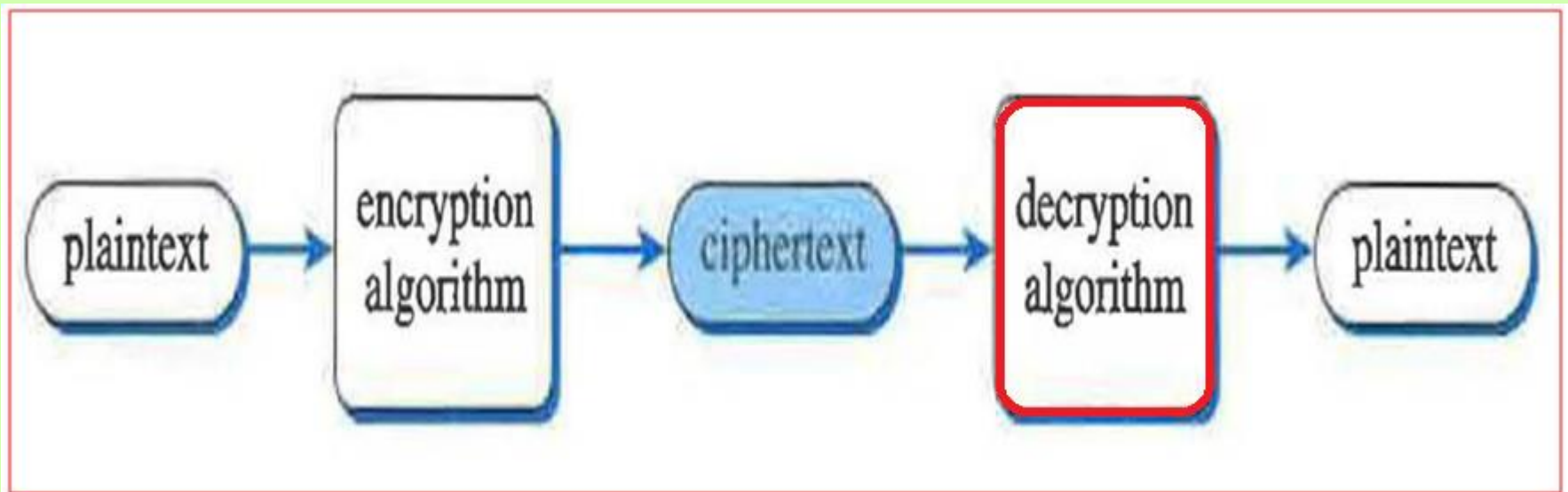
- The `scramble2Encrypt` function makes use of the accumulator pattern in several different places.
- The order of variables in the concatenation is not accidental.

Encrypting Using Transposition (cont'd.)

- Notice that some of the test cases are just nonsense strings, but they are chosen carefully to make it easy to see if the function is working the way we think it should.
- Notice that we test some boundary cases like a string of length 1 and even an empty string.

Decrypting Messages

- Write a function to decrypt a message that was encrypted by `scramble2Encrypt` function.



Decrypting a Transposed Message

- To restore the plaintext string, we start out by **splitting** the ciphertext in half.
- The first half of the string contains the odd characters from our original message, and the second half of the string contains the even characters.

Decrypting a Transposed Message (cont'd.)

I_a__akadsom_ih

0 2 4

It was

1 3

twsad_r_n_tryngt

The diagram illustrates the decryption of a transposed message. The ciphertext 'I_a__akadsom_ih' is shown at the top. Below it, the plaintext 'It was' is shown. Arrows indicate the mapping from the ciphertext to the plaintext. The first three columns of the ciphertext are mapped to the first three characters of the plaintext: 'I' to 'I', 'a' to 't', and 'a' to 'w'. The next three columns are mapped to the next three characters: 'k' to 'a', 'a' to 's', and 'd' to ' '. The final three columns are mapped to the final three characters: 's' to ' ', 'o' to ' ', and 'm' to ' '. The plaintext 'twsad_r_n_tryngt' is shown at the bottom, with arrows indicating the mapping from the ciphertext to the plaintext: 't' to 'I', 'w' to 't', 's' to 'a', 'a' to 's', 'd' to ' ', 'r' to ' ', 'n' to ' ', 't' to ' ', 'r' to ' ', 'y' to ' ', 'n' to ' ', 'g' to ' ', and 't' to ' '.

Decrypting a Transposed Message (cont'd.)

- One detail to consider when reconstructing the plaintext message is that we **may have one more character in the even character string than we do in the odd character string.**
- We can easily **check** for this by comparing the lengths of the two strings.
- If the odd-numbered character string is shorter than the even, we simply **concatenate the last character from the even string onto the plaintext.**

Decrypting a Transposed Message (cont'd.)

```
def scramble2Decrypt(cipherText):  
    halfLength = len(cipherText) // 2  
    oddChars = cipherText[:halfLength]  
    evenChars = cipherText[halfLength:]  
    plainText = ""  
  
    for i in range(halfLength):  
        plainText = plainText + evenChars[i]  
        plainText = plainText + oddChars[i]  
  
    if len(oddChars) < len(evenChars):  
        plainText = plainText + evenChars[-1]  
  
    return plainText
```

Decrypting a Transposed Message (cont'd.)

```
>>> scramble2Encrypt('abababc')
'bbbaaac'
>>> scramble2Decrypt('bbbaaac')
'abababc'
>>> scramble2Decrypt(scramble2Encrypt('abababc'))
'abababc'
>>> scramble2Decrypt(scramble2Encrypt('a'))
'a'
>>> scramble2Decrypt(scramble2Encrypt(' '))
' '
>>> scramble2Encrypt('I do not like green eggs and ham')
' ontlk re gsadhmId o iegeneg n a'
>>> scramble2Decrypt(' ontlk re gsadhmId o iegeneg n a')
'I do not like green eggs and ham'
>>> scramble2Decrypt(scramble2Encrypt('I do not like green eggs and ham'))
'I do not like green eggs and ham'
```

Mapping from Characters to Numbers and Back in

- For our purposes, we want the letter 'a' to map to the number 0 and the letter 'z' to map to the number 25.
- We define our own helper functions for mapping from characters to numbers and back in.

0	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15	16	17	18	19	20	21	22	23	24	25
a	b	c	d	e	f	g	h	i	j	k	l	m	n	o	p	q	r	s	t	u	v	w	x	y	z

```

def letterToIndex(ch) :
    alphabet = "abcdefghijklmnopqrstuvwxyz "
    idx = alphabet.find(ch)
    if idx < 0:
        print ("error: letter not in the alphabet", ch)
    return idx

def indexToLetter(idx) :
    alphabet = "abcdefghijklmnopqrstuvwxyz "
    if idx > 26:
        print ('error: ', idx, ' is too large')
        letter = ''
    elif idx < 0:
        print ('error: ', idx, ' is less than 0')
        letter = ''
    else:
        letter = alphabet[idx]
    return letter

```

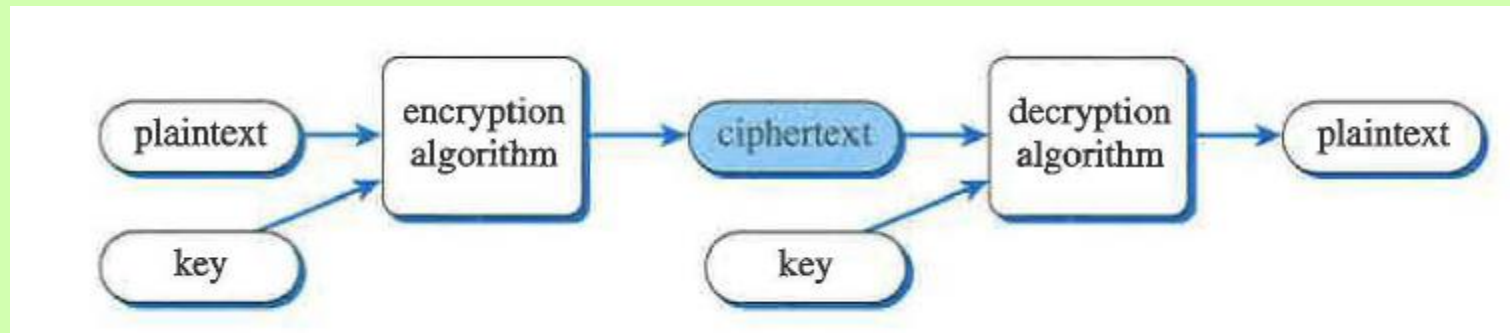
```

>>> letterToIndex('c')
2
>>> letterToIndex('?')
error: letter not in the alphabet ?
-1
>>> indexToLetter(3)
'd'
>>> indexToLetter(27)
error:  27  is too large
''
>>> indexToLetter(-1)
error:  -1  is less than 0
''

```

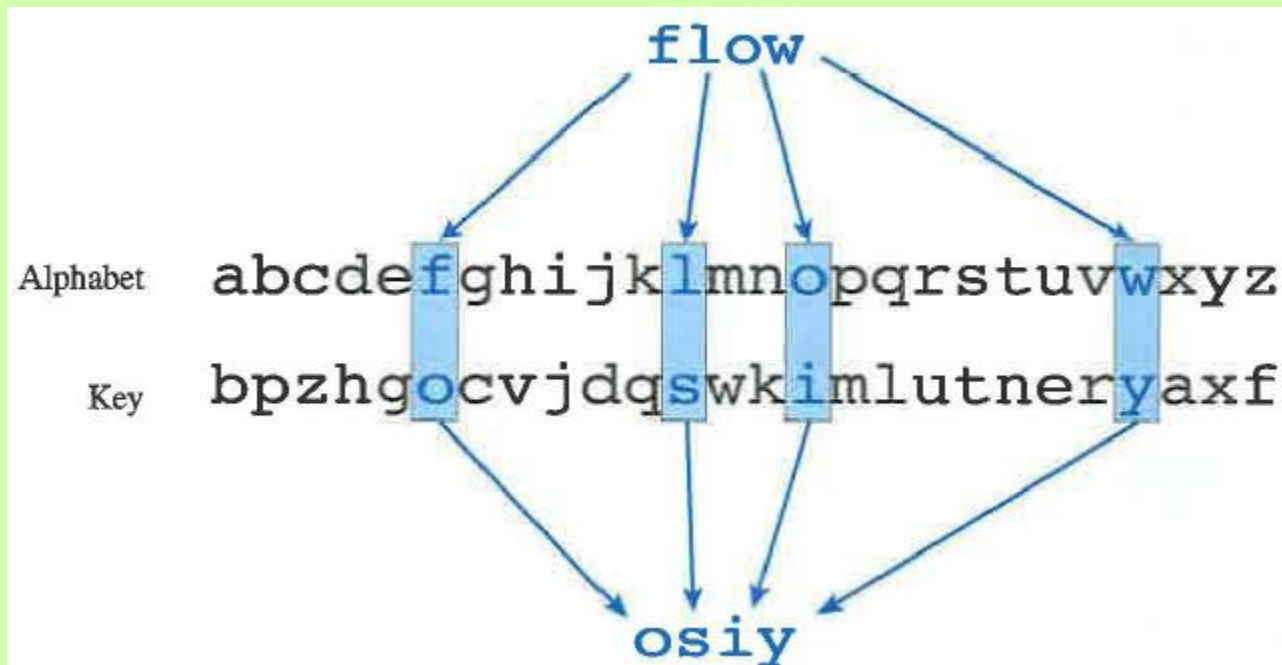
Substitution Cipher

- The substitution cipher substitutes one letter for another throughout a message.
- The substitution cipher has one big advantage over the transposition cipher: It uses a key.
- The key is created by rearranging the alphabet.
- If we just consider the 26 letters of the alphabet, there are 26 factorial different possible rearrangements of the alphabet.



Substitution Cipher (cont'd.)

- The important step in the encryption process is to take a letter from the plaintext alphabet and map it into a letter in the ciphertext alphabet.



Substitution Cipher (cont'd.)

```
def substitutionEncrypt(plainText, key):  
    alphabet = "abcdefghijklmnopqrstuvwxyz "  
    plainText = plainText.lower()  
    cipherText = ""  
    for ch in plainText:  
        idx = alphabet.find(ch)  
        cipherText = cipherText + key[idx]  
    return cipherText  
  
testKey1 = "zyxwvutsrqponmlkjihgfedcba "  
testKey2 = "ouwckbjmpzyexavrltsfgdqihn "  
cipherText = substitutionEncrypt("the quick brown fox", testKey1)  
print(cipherText)  
cipherText = substitutionEncrypt("the quick brown fox", testKey2)  
print(cipherText)  
  
gsv jfrxp yildm ulc  
fmk lgpwy utvqa bvi
```