Loop patterns

Practice 04.03

By Marina Barsky

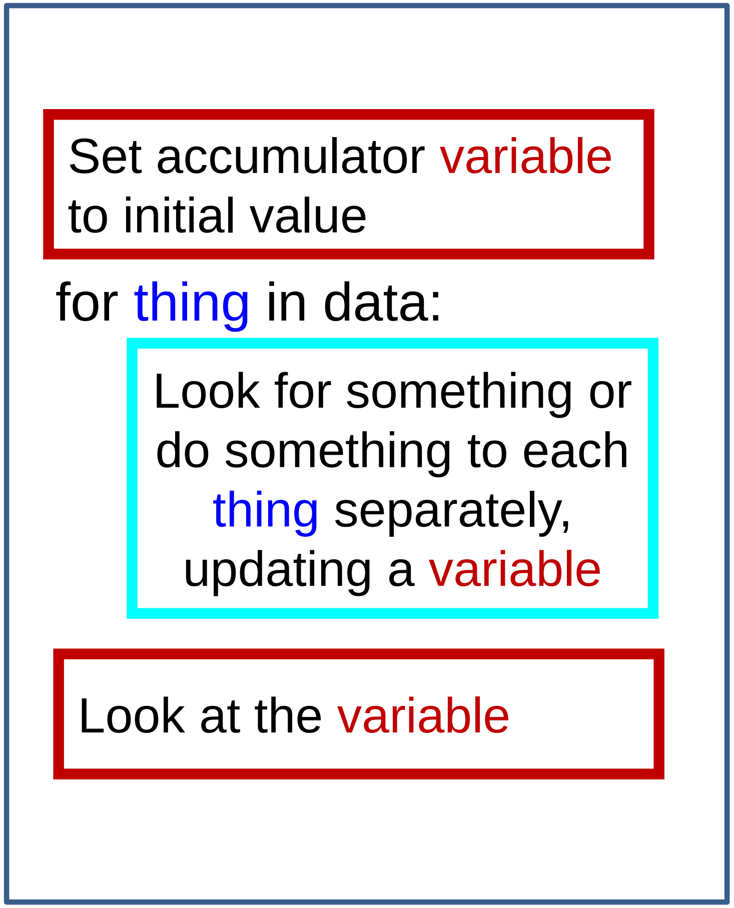
Loop Idioms:
what we do with loops

Note: Even though these examples are simple, the patterns apply to all kinds of loops

1. Accumulators
2. Parallel lists and loops over indices
3. Nested loops

1. The main pattern of `for` loops: accumulators

- Accumulator variable initialized outside the loop
- The variable accumulates some value in the body of the loop using iteration variable
- When we done with the loop: output the value accumulated in the variable



Set accumulator variable to initial value

for thing in data:

Look for something or do something to each thing separately, updating a variable

Look at the variable

What is the Largest Number?

3 41 12 9 74 15

3

```
for n in a_list:
    n ?
```

for loops: max val in the list

```
largest = None
print('Before:', largest)
for iterval in [3, 41, 12, 9, 74, 15]:
    if largest is None or largest < iterval:
        largest = iterval
    print('Loop:', iterval, largest)
print('Largest:', largest)
```

for loops: min val in the list

```
def find_min( a_list ):
   min = None
   for x in a_list:
        if min is None or min > x:
           min = x
   return min
```

2. Looping through parallel lists

- If we need to iterate over elements of ***more than one list at the same time*** we use **`for`** loops of type II: loops over indices
- If we traverse several lists at the same time, we say that we are working with parallel lists (or strings)

- *Example*:

Given two strings of the same length, count how many times the characters at the same position differ (humming distance)

Parallel min difference

- Given two lists of numbers, compute the minimum difference among any pair of elements **at the same position** in both lists.
- E.g., list1 = [1, 2, 3, **1**], list2 = [-2, 10, 5, **0**, 6], the function min_diff would return 1, which is the difference for position 3 in both lists: {1,0}

- The ideas are similar to the *find_min*, only in the loop we iterate over both lists – that means we need to iterate over indices, not elements

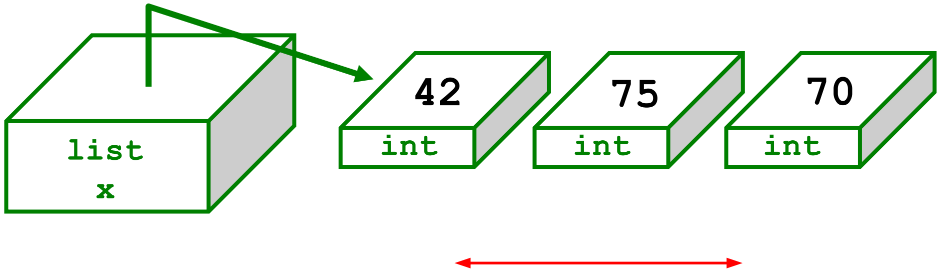
Parallel min difference: solution

```
def min_diff( list1, list2 ):
   '''
   The parameters list1, list2 are two int lists.
   Find minimum difference among any pairs at the
   same position.
   '''
   min_sofar = None
   min_len = min(len(list1),len(list2))
   for i in range(min_len):
      diff = abs(list1[i] - list2[i])
      if min_sofar is None or diff < min_sofar:
          min_sofar = diff
   return min_sofar
```

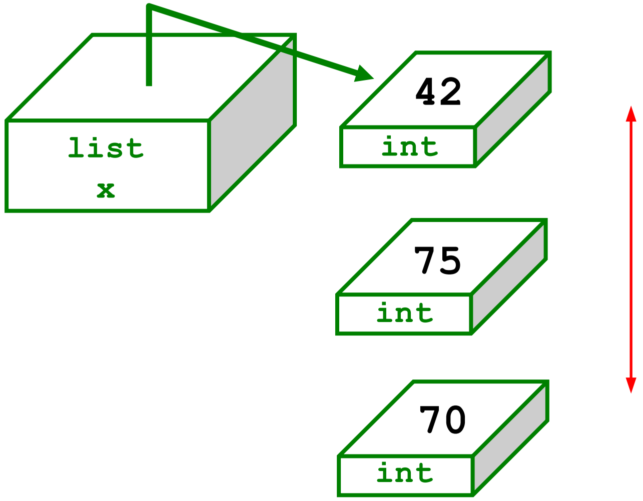
2. Nested lists and nested loops

Lists can hold **ANY** type of data

```
x = [ 42, 75, 70 ]
```

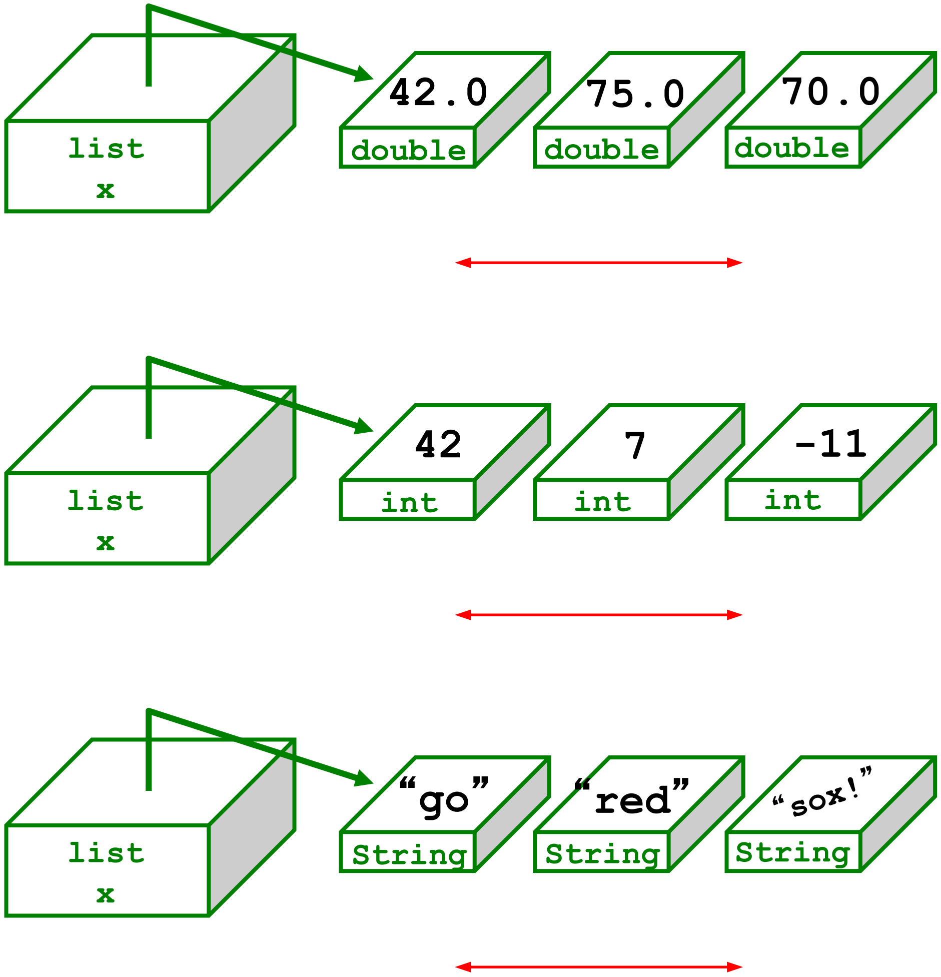


We can equally well imagine them as **vertical** structures.



List elements can be numbers or strings

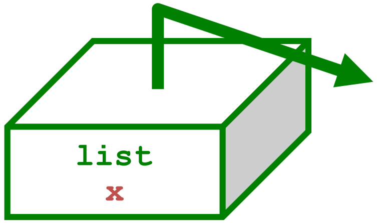
Lists can hold **ANY** type of data



2D lists

Lists can hold **ANY** type of data -- including lists !

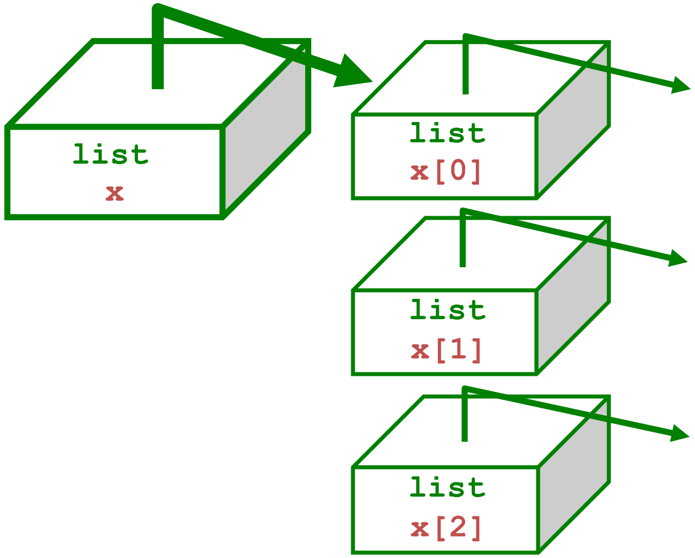
```
x = [ [1,2,3,4], [5,6], [7,8,9,10,11] ]
```



2D lists

Lists can hold **ANY** type of data -- including lists !

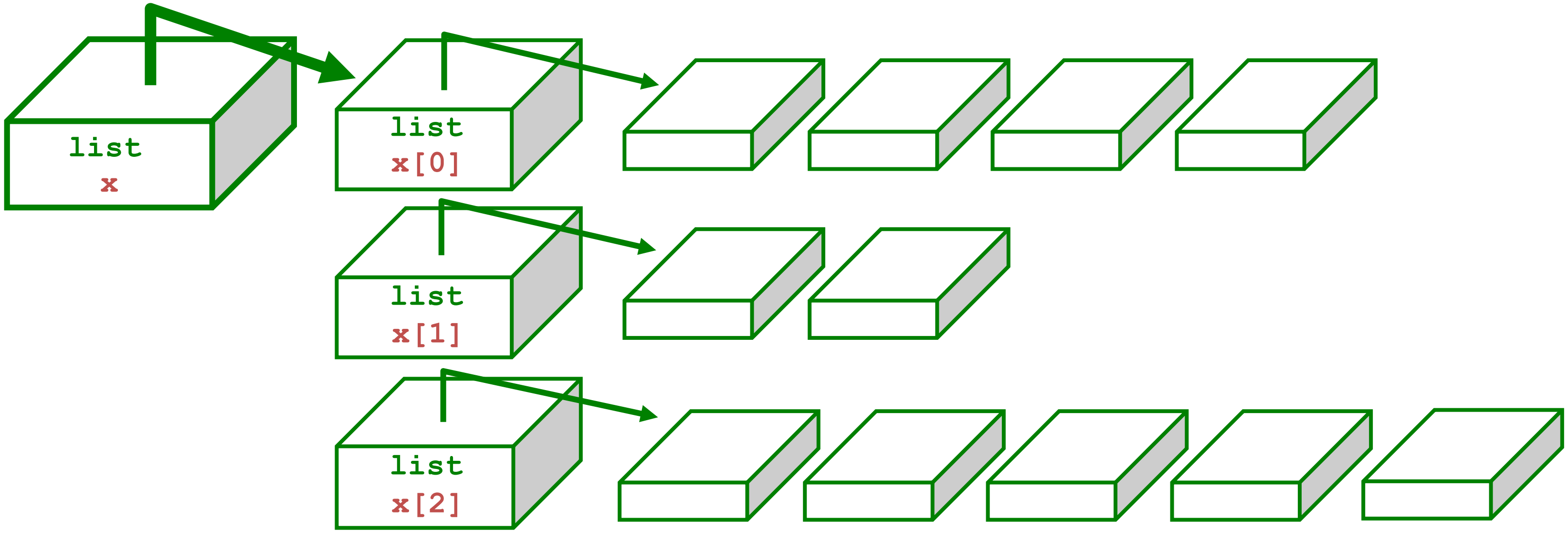
```
x = [ [1,2,3,4], [5,6], [7,8,9,10,11] ]
```



Jagged lists

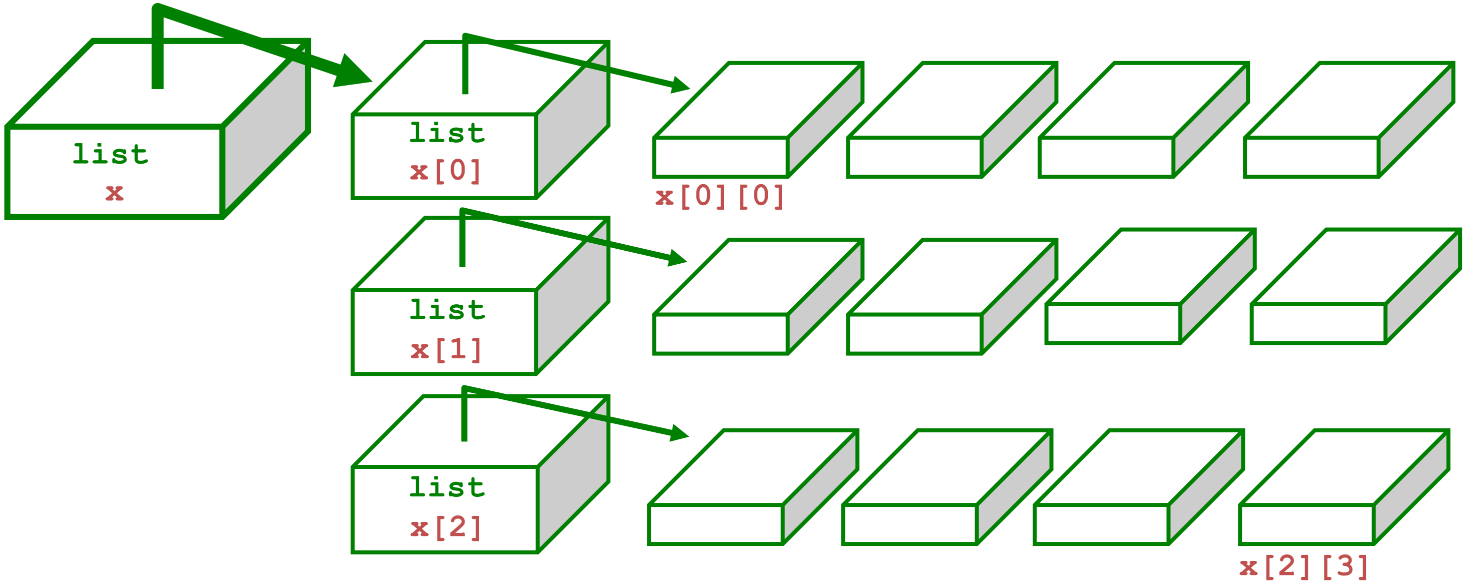
Lists can hold **ANY** type of data -- including lists !

```
x = [ [1,2,3,4], [5,6], [7,8,9,10,11] ]
```



Rows within 2d lists ***need not be the same length***

Rectangular lists



What does `x[1]` refer to?

What value is changed with `x[1][2]=42` ?

How many rows does `x` have, in general ?

How many columns does `x` have, in general ?

Concrete example

```
grades = [['Assignment 1', 80],
          ['Assignment 2', 90],
          ['Assignment 3', 70]]
```

```
sublist = grades[0]
sublist[0]
sublist[1]

grades[0][0]
grades[1][0]
grades[2][1]
```

Number of rows in this table?
Number of columns?

Nested loops

- The bodies of loops can contain any statements, **including other loops**. When this occurs, this is known as a ***nested loop***.

- In this case we have **more than one iteration variable**:

```
num_list = [1, 2, 3]
alpha_list = ['a', 'b', 'c']
for number in num_list:
    print(number)
    for letter in alpha_list:
        print(letter)
```

Here *number* is an iteration variable in the ***outer*** loop, and for each value of number – there is another ***inner*** loop with its own iteration variable *letter*

Nested loops: finger exercise

```
num_list = [1, 2, 3]
alpha_list = ['a', 'b', 'c']
for number in num_list:
    print(number)
    for letter in alpha_list:
        print(letter)
```

number	**letter**	output
1		1
1	a	a
1	b	b
1	c	c
2		2
2	a	a
2	b	b
2	c	c
3		3
3	a	a
3	b	b
3	c	c

Example 1. Analyze

- What is printed here?

```
for i in range(10, 13):
    for j in range(1, 3):
        print(i, j)
```

Example 2. Analyze

```
list_of_lists = [ ['uno', 'dos’],
                  [1, 2],
                  ['one', 'two', 'three']]

for list in list_of_lists:
    print(list)

for list in list_of_lists:
    for item in list:
        print(item)
```

Example 3. Analyze

```
names=['ann','ali', 'bob']
cars=['mercedes','porshe']
numbers=[1,2,3]
for name in names:
  for car in cars:
    for number in numbers:
      print("{0} has {1} of {2}".format(
                  name,number,car))
```

Example 4. Program

- Given two lists of numbers, compute the minimum difference among any pair of numbers, one from each list.
- E.g., list1 = [1, 2, 3, 4], list2 = [-2, 10, 5, 0, 6], the function *min_diff_all* would return 1, which occurs twice, {1,0}, {4,5}.

The ideas are similar to the *find_min*, only this time we need iterate over all possible value combinations in two lists

Solution: total min difference

```
def min_diff_all( list1, list2 ):
   '''
   The parameters list1, list2 are
   two int lists.
   Find minimum difference among any pairs.
   '''
   min_sofar = None
   for x in list1:
      for y in list2:
          diff = abs(x - y)
          if diff is None or diff < min_sofar:
             min_sofar = diff
   return min_sofar
```

Example 5. Program

- Given *num_rows* and *num_cols*, print a list of all seats in a theater. Rows are numbered, columns lettered, as in 1A or 3E.
- Print a space after each seat, including after the last.
- Use separate print statements to print the row and column. Ex: *num_rows* = 2 and *num_cols* = 3 prints:

```
1A 1B 1C 2A 2B 2C
```

Optional parameter to *print* – tells what to do when the line ends

```
print('A', end=' ')
print()
```

Moves to the next line

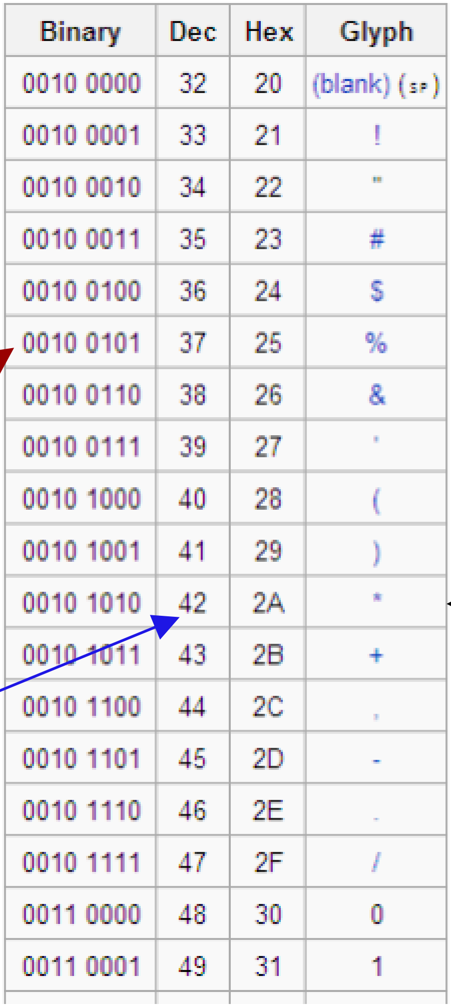
ASCII

American Standard Code for Information Interchange

ASCII is a table that tells the computer how to represent characters as bits!

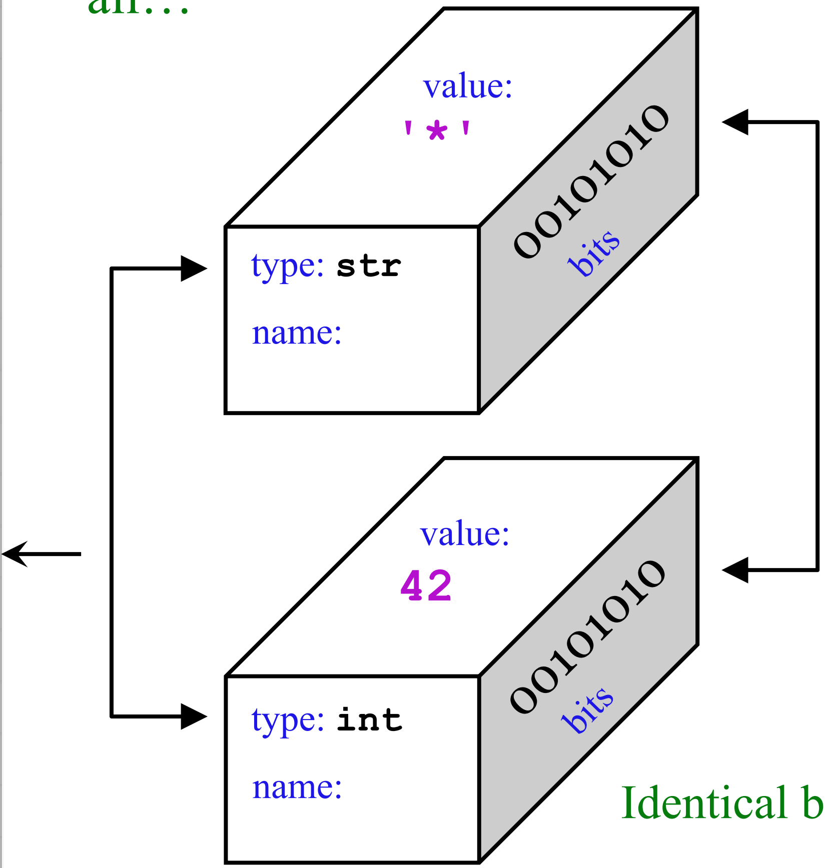
8 bits = 1 byte

The SAME bits represent integers, if the variable has type **int** instead of **str**



Binary	Dec	Hex	Glyph
0010 0000	32	20	(blank) (SP)
0010 0001	33	21	!
0010 0010	34	22	"
0010 0011	35	23	#
0010 0100	36	24	$
0010 0101	37	25	%
0010 0110	38	26	&
0010 0111	39	27	'
0010 1000	40	28	(
0010 1001	41	29	)
0010 1010	42	2A	*
0010 1011	43	2B	+
0010 1100	44	2C	,
0010 1101	45	2D	-
0010 1110	46	2E	.
0010 1111	47	2F	/
0011 0000	48	30	0
0011 0001	49	31	1

The types determine how to interpret the bits; the names don't matter at all…



Identical bits!

ASCII

Converting between numbers and characters

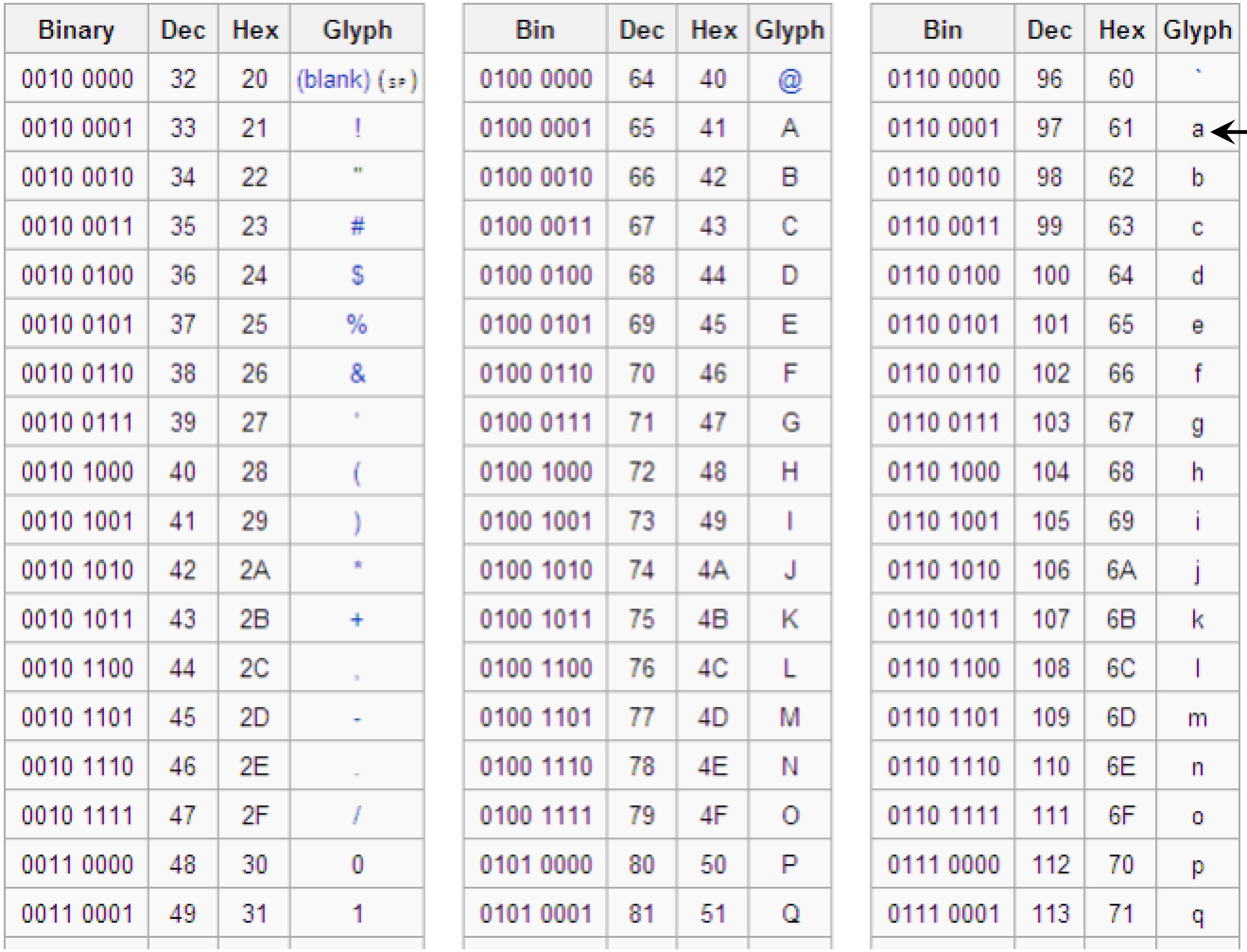
ASCII is a table that tells the computer how to represent characters as #s

`chr(97)` is `'a'`

`chr` convert number to character.

`ord` convert character to number

`ord('a')` is 97



Binary	Dec	Hex	Glyph
0010 0000	32	20	(blank) (␠)
0010 0001	33	21	!
0010 0010	34	22	"
0010 0011	35	23	#
0010 0100	36	24	$
0010 0101	37	25	%
0010 0110	38	26	&
0010 0111	39	27	'
0010 1000	40	28	(
0010 1001	41	29	)
0010 1010	42	2A	*
0010 1011	43	2B	+
0010 1100	44	2C	,
0010 1101	45	2D	-
0010 1110	46	2E	.
0010 1111	47	2F	/
0011 0000	48	30	0
0011 0001	49	31	1

Bin	Dec	Hex	Glyph
0100 0000	64	40	@
0100 0001	65	41	A
0100 0010	66	42	B
0100 0011	67	43	C
0100 0100	68	44	D
0100 0101	69	45	E
0100 0110	70	46	F
0100 0111	71	47	G
0100 1000	72	48	H
0100 1001	73	49	I
0100 1010	74	4A	J
0100 1011	75	4B	K
0100 1100	76	4C	L
0100 1101	77	4D	M
0100 1110	78	4E	N
0100 1111	79	4F	O
0101 0000	80	50	P
0101 0001	81	51	Q

Bin	Dec	Hex	Glyph
0110 0000	96	60	`
0110 0001	97	61	a
0110 0010	98	62	b
0110 0011	99	63	c
0110 0100	100	64	d
0110 0101	101	65	e
0110 0110	102	66	f
0110 0111	103	67	g
0110 1000	104	68	h
0110 1001	105	69	i
0110 1010	106	6A	j
0110 1011	107	6B	k
0110 1100	108	6C	l
0110 1101	109	6D	m
0110 1110	110	6E	n
0110 1111	111	6F	o
0111 0000	112	70	p
0111 0001	113	71	q

`chr` and `ord`

ASCII VALUES

`abcdefghijklmnopqrstuvwxyz`

97 99 101 103 105 107 109 111 113 115 117 119 122

`ABCDEFGHIJKLMNOPQRSTUVWXYZ`

65 67 69 71 73 75 77 79 81 83 85 87 90

CONVERTERS

`ord(c)`

Input: a string of one character, **`c`**
Output: an integer, the ASCII value of c

`chr(n)`

Input: an integer in **`range(256)`**
Output: a one-char. string of that ASCII value

try these!

```
for i in range(128):
   print(i,chr(i))

for i in '**** CS! ****’:
   print(ord(i))
```

`chr` and `ord`

ASCII

VALUES

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
97		99		101		103		105		107		109		111		113		115		117		119			122
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
65		67		69		71		73		75		77		79		81		83		85		87			90

`ord('a')` is `97`

`chr(66)` is `'B'`

What is `chr(ord('i')+3)`?

What is `chr(ord('Y')+3)`?

Solution: theater seats

```
def print_seats( num_rows, num_cols ):
   first_seat = ord('A')
   for i in range(1, num_rows+1):
      for j in range(first_seat, first_seat+num_cols):
          seat = chr (j)
          print (i, end='')
          print (seat, end=' ')
```

Example 6. Program

- Write a function that given a list of strings returns the string with the largest number of vowels
- For example for list t = ['africa', 'america', 'Australia'] returns 'Australia'.

Solution: most vowels

```
def most_vowels( t ):
    max_sofar = None
    best_index = 0
    for i in range(len(t)):
        s = t[i]  #looking at the current string
        count = 0
        for c in s:
            if c in 'aeiou':
                count += 1

        if max_sofar is None or count > max_so_far:
            max_sofar = count
            best_index = i
    return t [best_index]
```

Inner loop for counting vowels in s

Nested loops for printing patterns

```
for row in range(3):
  print('# # # # ')
```

output?

Patterns

```
for row in range(3):
    print('# # # # ')
```

```
# # # #
# # # #
# # # #
```

Not particularly flexible!

Patterns

```
for row in range(3):
  for col in range(4):
    print('#')
```

```
# # # #
# # # #
# # # #
```

Is this still the output?

NO! What changes are needed?

Nested loops are powerful – and flexible...

Tracking rows and columns

```
for row in range(3):
    for col in range(4):
        print('$', end='')
    print()
```

	0	1	2	3	cols
0					
1					
2					
rows					

Pattern 1

Change each block of code so that it will print the examples below:

```
for row in range( 3 ):
 for col in range( 6 ):
  print(_____)
 print()
```

```
0 1 2 3 4 5
0 1 2 3 4 5
0 1 2 3 4 5
```

General approach

```
0 1 2 3 4 5
0 1 2 3 4 5
0 1 2 3 4 5
```

- We must build multiple lines of output using:
  - an outer "vertical" loop for each of the lines
  - inner "horizontal" loop(s) for the patterns within each line

Step 1

```
0 1 2 3 4 5
0 1 2 3 4 5
0 1 2 3 4 5
```

- First write the outer loop which iterates specified number of rows and moves to the next row with each iteration

```
for row in range( 3 ):

    print()
```

Step 2

```
0 1 2 3 4 5
0 1 2 3 4 5
0 1 2 3 4 5
```

- Now look at the line contents. Each line has a ***pattern***.
- In this case each line has the same 6 numbers from 0 to 5

```
for row in range( 3 ):
  for col in range( 6 ):
    print(col, end=' ')
  print()
```

Pattern 2

Change each block of code so that it will print the examples below:

```
for row in range( 3 ):
  for col in range( 6 ):
    print(__________)
  print()
```

```
0 0 0 0 0 0
1 1 1 1 1 1
2 2 2 2 2 2
```

Pattern 2 solution

Change each block of code so that it will print the examples below:

```
for row in range( 3 ):
  for col in range( 6 ):
    print(row,end=' ')
  print()
```

```
0 0 0 0 0 0
1 1 1 1 1 1
2 2 2 2 2 2
```

Pattern 3

Change each block of code so that it will print the examples below:

```
for row in range( 3 ):
  for col in range( 6 ):
    print(__________)
  print()
```

```
0 1 2 3 4 5
1 2 3 4 5 6
2 3 4 5 6 7
```

Pattern 3 solution

Change each block of code so that it will print the examples below:

```
for row in range( 3 ):
  for col in range( 6 ):
    print(col+row,end=' ')
  print()
```

```
0 1 2 3 4 5
1 2 3 4 5 6
2 3 4 5 6 7
```

Pattern 4

Change each block of code so that it will print the examples below:

```
for row in range( 3 ):
  for col in range( 6 ):
    print(__________)
  print()
```

```
0 1 0 1 0 1
1 0 1 0 1 0
0 1 0 1 0 1
```

Pattern 4 solution

Change each block of code so that it will print the examples below:

```
for row in range( 3 ):
  for col in range( 6 ):
    print((col+row)%2,end=' ')
  print()
```

```
0 1 0 1 0 1
1 0 1 0 1 0
0 1 0 1 0 1
```

Self-exercises

- **Important limitation!** For these problems, you should **not** use Python’s string-multiplication or string-addition operators. Because our goal is to use loop constructs, use loops to achieve the repetition that those operators might otherwise provide. There is one exception, however — you ***may*** use string-multiplication with the space character ' '. That is, you can create any number of consecutive spaces with constructs like **' '*n**

Problem 1. print_rect

- Write a function named *print_rect* that takes three arguments: *width*, *height*, and *symbol*, and prints a width by height rectangle of symbols on the screen.

```
>>> print_rect(4, 6, '%')
% % % %
% % % %
% % % %
% % % %
% % % %
% % % %
```

Problem 2. print_triangle

- Create a function *print_triangle* that takes three arguments: *leg*, *symbol*, and *right_side_up* and prints a right-angled triangle of symbols on the screen. *leg* is a number that determines the width of the sides of the triangle forming the right angle, and *right_side_up* is a boolean that determines whether the triangle is printed right side up (True) or upside down (False).

```
>>> print_triangle(3, '@', False)
@
@ @
@ @ @

>>> print_triangle(3, '@', True)
@ @ @
@ @
@
```

Problem 3. print_bumps

- Now, use your *print_triangle* function to write a function called *print_bumps (num, symbol1, symbol2)* that will print the specified number of two-symbol "bumps", where each bump is larger than the last, as in the following example:

```
>>> print_bumps(4, '%', '#')
%
#
%
% %
# #
#
%
% %
% % %
# # #
# #
#
%
% %
% % %
% % % %
# # # #
# # #
# #
#
```

Problem 4. print_diamond

- Write a function called *print_diamond (width, symbol)* that prints a diamond of symbol whose maximum width is determined by *width*.

```
>>> print_diamond(3, '+')
  +
 + +
+ + +
 + +
  +
```

Problem 5. print_striped_diamond

- Next, write a function called *print_striped_diamond (width, symbol1, symbol2)* that prints a "striped diamond" of *symbol1* and *symbol2*. For example:

```
>>> print_striped_diamond (7, '.', '%')
      .
     . %
    . % .
   . % . %
  . % . % .
 . % . % . %
. % . % . % .
% . % . % .
 . % . % .
  % . % .
   . % .
    % .
     .
```

challenge

Problem 6. print_crazy_striped_diamond

- Finally, write a function called *print_crazy_striped_diamond (width, symbol1, symbol2, symbol1_width, symbol2_width)* that prints a “striped diamond” of *symbol1* and *symbol2* where the stripes can have varied widths: *symbol1_width* determines the width of the stripe made of *symbol1* and *symbol2_width* determines the width of the stripe made of *symbol2*.
- For example:

```
>>> print_crazy_striped_diamond (7, '.', '%', 2, 1)
      .
     . .
    . . %
   . . % .
  . . % . .
 . . % . . %
. . % . . % .
 . % . . % .
  % . . % .
   . . % .
    . % .
     % .
      .
```