

Discussion questions #3

String, Data structure (Data abstraction)

Q1:What is Abstraction? Why do we need it?

What are the two aspects of Abstraction?

- Abstraction is a process to identify what is important without worrying too much about the detail so that we can focus on the important part.
- Since Abstraction provides a means to distil what is essential, it can help us to come up with manageable approach to create computational solutions. In addition, Abstraction also help us to view things at different degrees of detail.
- Abstraction can be performed in two main aspects: data and algorithm, which form the two bases for programming: data structures and functions. This session is focus on the first part, data structures.



DATA STRUCTRES



Q2:

- When choosing a password for online accounts, there are typically certain requirements for the strength of the password. Develop a Python program for testing if a string satisfies some appropriate criteria for a strong password. It's up to you to define the requirements.

- The length of the password is more than 8 characters
- At least one upper case letter
- At least one lower case letter
- At least one digit
- At least one special symbol (punctuation)

<https://docs.python.org/3/library/stdtypes.html#text-sequence-type-str>

`len(str)`

`str.isupper()`

`str.islower()`

`str.isdigit()`

```
LENGTH = 8          # the minimum length for a strong password

password = input("Input your password: ")

upCase = False      # indicating if the password has at least one
lowCase = False     # indicating if the password has at least one
digit = False       # indicating if the password has at least one

for char in password:    # iterate on each character of the pas

    if char.isupper():    # if the character is in upper case, s
        upCase = True

    if char.islower():    # if the character is in lower case, s
        lowCase = True

    if char.isdigit():    # if the character is a digit, set dig
        digit = True

length = len(password)   # get the length of the password

strong = upCase and lowCase and digit and length > LENGTH
# strong would be True, if all the conditions hold

if strong:
    print("Your password is strong enough.")
else:
    print("Your password is weak.")
```

str.ispunctuation ()

```
import string
for char in password:
    if char in string.punctuation:
        print("punctuation = True")
```



Free Photoshop PSD file download - ResoJu

6.1. `string` — Common string operations

Source code: [Lib/string.py](#)

See also: [Text Sequence Type — str](#)

[String Methods](#)

6.1.1. String constants

The constants defined in this module are:

`string.ascii_letters`

The concatenation of the `ascii_lowercase` and `ascii_uppercase` constants described below. This value is not locale-dependent.

`string.ascii_lowercase`

The lowercase letters `'abcdefghijklmnopqrstuvwxyz'`. This value is not locale-dependent and will not change.

`string.ascii_uppercase`

The uppercase letters `'ABCDEFGHIJKLMNOPQRSTUVWXYZ'`. This value is not locale-dependent and will not change.

`string.digits`

The string `'0123456789'`.

`string.hexdigits`

The string `'0123456789abcdefABCDEF'`.

`string.octdigits`

The string `'01234567'`.

`string.punctuation`

String of ASCII characters which are considered punctuation characters in the `C` locale.

Python List is an **ordered sequence of items**.



Recall

We have already covered a type of sequence: **Strings**

- A **string** is a **sequence of characters**.

dad \neq add

[255, 0, 0] \neq [0, 0, 255]

Lists: Differences with Strings

- Lists can contain **a mixture of python objects (types)**; strings can **only hold characters**.

E.g. `l = [1, 'bill', 1.2345, True]`

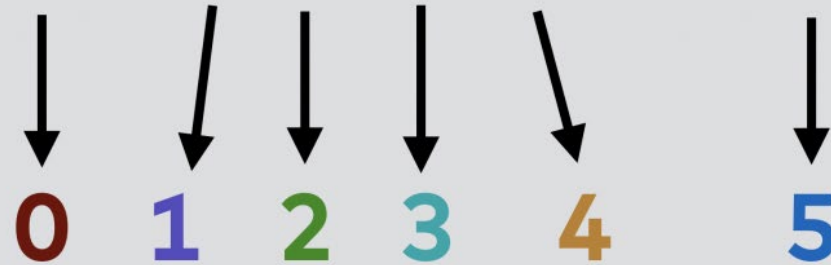
- Lists are **mutable**; their values can be changed, while strings are **immutable**.
- Lists are designated with `[]`, with elements separated by commas; strings use `""`.

List vs Dictionary

```
dog_name = 'Freddie'  
age = 9  
is_vaccinated = True  
height = 1.1  
birth_year = 2001
```

Freddie has two belongings: a bone and a little ball.

```
dog = ['Freddie', 9, True, 1.1, 2001, ['bone', 'little ball']]
```



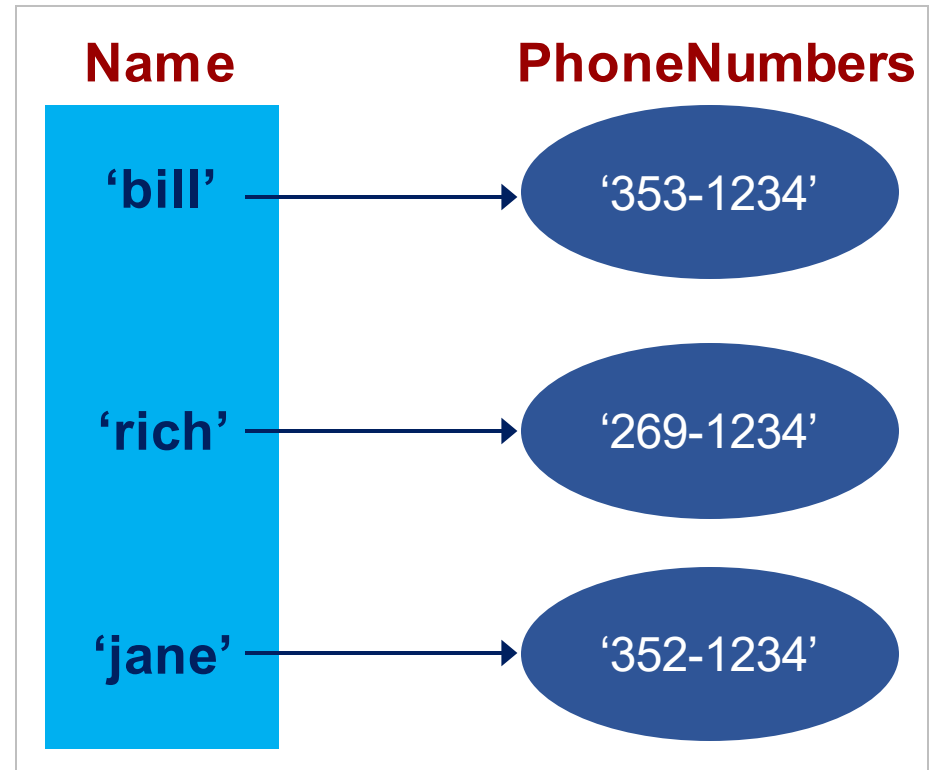
```
dog_dict = {'name': 'Freddie', 'age': 9, 'is_vaccinated': True, 'height': 1.1, 'birth_year': 2001, 'belongings': ['bone', 'little ball']}
```

In a dictionary you can attribute a unique key for each of these values, so you can understand better that what value stands for what.

{ } **marker**: used to create a dictionary

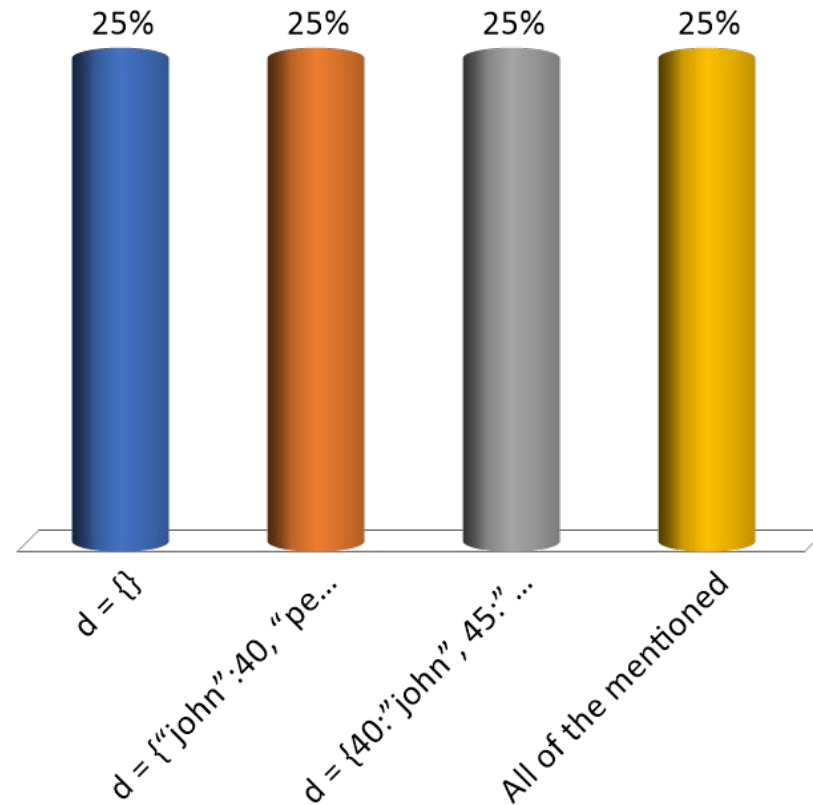
: **marker**: used to create **key:value** pairs

```
contacts = {'bill': '353-1234',  
            'rich': '269-1234',  
            'jane': '352-1234'}  
  
print(contacts) ➔ {'jane': '352-1234',  
                   'bill': '353-1234',  
                   'rich': '269-1234'}
```



Which of the following statements create a dictionary?

- A. `d = {}`
- B. `d = {"john":40, "peter":45}`
- C. `d = {40:"john", 45:"peter"}`
- ✓ D. All of the mentioned

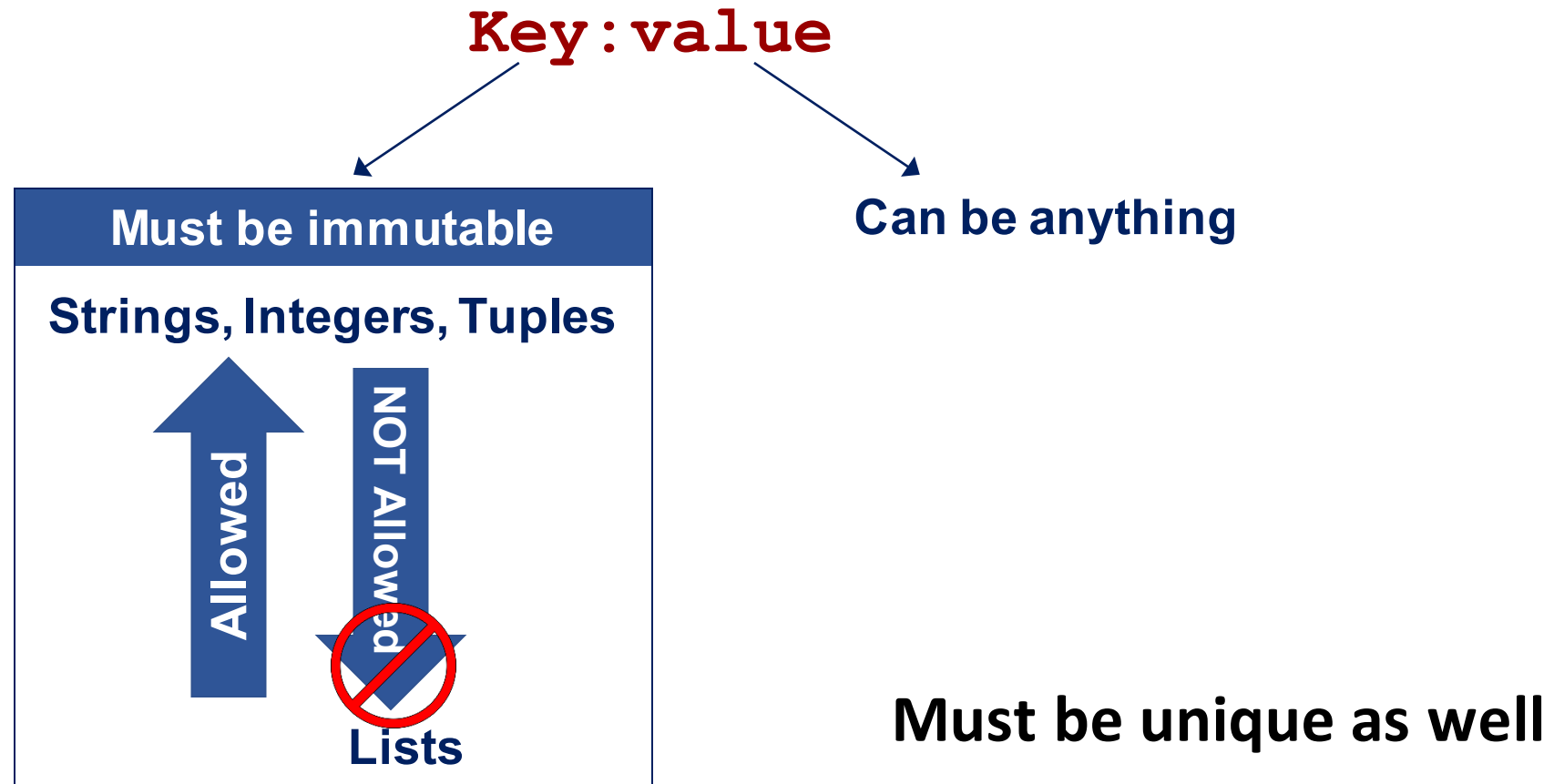


Q3. grade of each student in each lab class

- Consider a system for storing anonymous grades of each lab class. Define a data structure, which can identify individuals in each lab group by an ID number 1-40 (inclusive). To identify the person in the entire class you would also need the group name, e.g., 'FE2'. Each corresponding person should have a number between 1-100 (inclusive) to define grade.

Group	Student ID	Grade
FS1	1	45
FS1	2	75
FS1	3	25
FS1	4	65
FS2	1	85
FS2	2	40
FS2	3	70
FS2	4	80
FS3	1	80
FS3	2	70
FS3	3	45
FS3	4	60

What are Keys and Values?



Tuples (,)

Tuples are **immutable** lists.

Why Immutable Lists?

- Provides a data structure with some integrity and some permanency
- To avoid accidentally changing one

They are designated with **(,)**.

Example:

```
myTuple = (1, 'a', 3.14, True)
```

Lists vs. Tuples

Everything that works for a list works for a tuple **except** methods that modify the tuple.

What works?

- indexing
- slicing
- `len()`
- `print()`

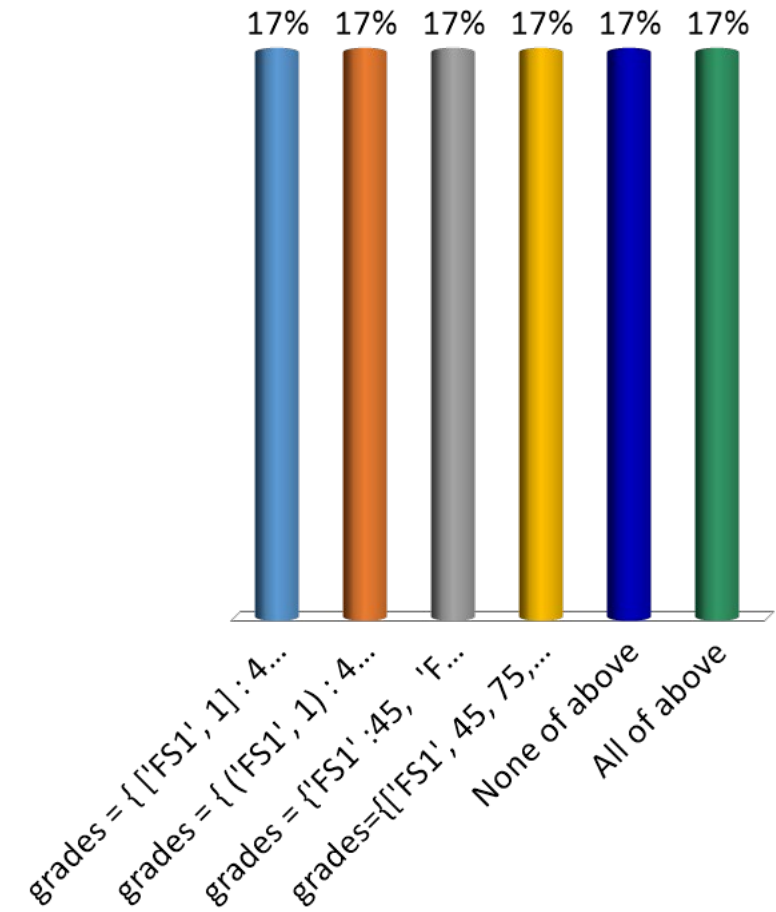
What doesn't work?

Mutable methods

- `append()`
- `extend()`
- `remove()` , etc.

Which of the following is an appropriate data structure for Q3?

- A. `grades = { ['FS1', 1] : 45, ['FS1', 2] : 75, ('FS2', 1) : 75 }`
- ✓ B. `grades = { ('FS1', 1) : 45, ('FS1', 2) : 75, ('FS2', 1) : 75 }`
- C. `grades = {'FS1' : 45, 'FS1' : 75, 'FS2' : 75 }`
- D. `grades = { ['FS1', 45, 75, 25, 65], ['FS2', 75, 40, 70, 80] }`
- E. None of above
- F. All of above



```
# solution : to use a dictionary
# the key is a tuple (group_name, ID). NOTICE:
# the key has to be immutable and unique
# the value is the grade
```

```
grades = {
    ('FS1', 1) : 45,
    ('FS1', 2) : 75,
    ('FS1', 3) : 25,
    ('FS1', 4) : 65,

    ('FS2', 1) : 75,
    ('FS2', 2) : 40,
    ('FS2', 3) : 70,
    ('FS2', 4) : 80
}

print(grades[('FS1', 1)])
```

Improvement: Create Database (sentinel : score = -1)

```
grades = {  
    ('FS1', 1) : 45,  
    ('FS1', 2) : 75,  
    ('FS1', 3) : 25,  
    ('FS1', 4) : 65,  
  
    ('FS2', 1) : 75,  
    ('FS2', 2) : 40,  
    ('FS2', 3) : 70,  
    ('FS2', 4) : 80  
}  
  
print(grades[('FS1', 1)])
```

```
grades = { } # the database is empty initially  
  
groupName = input("Please input the group name: ")  
sID = int(input("Please input the student id: "))  
score = int(input("Please input the score: "))  
while score != -1:  
    dataKey = (groupName, sID)  
    grades[dataKey] = score  
    groupName = input("Please input the group name: ")  
    sID = int(input("Please input the student id: "))  
    score = int(input("Please input the score: "))  
  
print(grades)
```

Basic List Operations

Lists respond to the `+` and `*` operators much like strings; they mean concatenation and repetition here too, except that the result is a new list, not a string. In fact, lists respond to all of the general sequence operations we used on strings.

Python Expression	Results	Description
<code>len([1, 2, 3])</code>	3	Length
<code>[1, 2, 3] + [4, 5, 6]</code>	<code>[1, 2, 3, 4, 5, 6]</code>	Concatenation
<code>['Hi!'] * 4</code>	<code>['Hi!', 'Hi!', 'Hi!', 'Hi!']</code>	Repetition
<code>3 in [1, 2, 3]</code>	True	Membership
<code>for x in [1, 2, 3]: print x,</code>	1 2 3	Iteration






Indexing, Slicing, and Matrixes










Because lists are sequences, indexing and slicing work the same way for lists as they do for strings.

```
L = ['spam', 'Spam', 'SPAM!']
```

Python Expression	Results	Description
L[2]	SPAM!	Offsets start at zero
L[-2]	Spam	Negative: count from the right
L[1:]	['Spam', 'SPAM!']	Slicing fetches sections

Built-in List Functions & Methods

Sr.No.	Function with Description
1	<code>cmp(list1, list2)</code>  Compares elements of both lists.
2	<code>len(list)</code>  Gives the total length of the list.
3	<code>max(list)</code>  Returns item from the list with max value.
4	<code>min(list)</code>  Returns item from the list with min value.
5	<code>list(seq)</code>  Converts a tuple into list.

Sr.No.	Methods with Description
1	<code>list.append(obj)</code>  Appends object obj to list
2	<code>list.count(obj)</code>  Returns count of how many times obj occurs in list
3	<code>list.extend(seq)</code>  Appends the contents of seq to list
4	<code>list.index(obj)</code>  Returns the lowest index in list that obj appears
5	<code>list.insert(index, obj)</code>  Inserts object obj into list at offset index
6	<code>list.pop(obj=list[-1])</code>  Removes and returns last object or obj from list
7	<code>list.remove(obj)</code>  Removes object obj from list
8	<code>list.reverse()</code>  Reverses objects of list in place
9	<code>list.sort([func])</code>  Sorts objects of list, use compare func if given

Q4:

- Given two lists of grades (list of integers) from two classes, write a Python program that will check which class has the highest average score and the highest maximum score.

Q4 grades

- Given two lists of grades (list of integers):

scores1 = [80, 91, 75, 80, 65, 76, 90, 84]

scores2 = [74, 59, 12, 98, 35, 42, 74, 75, 90]

- highest average score
- highest maximum score


```
scores1 = [80, 91, 75, 80, 65, 76, 90, 84]
```

```
scores2 = [74, 59, 12, 98, 35, 42, 74, 75, 90]
```

```
avg1 = float(sum(scores1) / len(scores1))
```

```
avg2 = float(sum(scores2) / len(scores2))
```

```
maxNum = max(max(scores1), max(scores2))
```

```
maxAvg = max(avg1, avg2)
```

```
print("Highest Avg: ", maxAvg)
```

```
print("Highest Score: ", maxNum)
```

List Comprehension

[expression **for**-clause *[condition]*]

List comprehension offers a shorter syntax when you want to create a new list based on the values of an existing list.

- *Example:*
- *Based on a list of fruits, you want a new list, containing only the fruits with the letter "a" in the name.*

```
fruits = ["apple", "banana", "cherry", "kiwi", "mango"]  
newlist = []
```

```
for x in fruits:  
    if "a" in x:  
        newlist.append(x)  
  
print(newlist)
```

```
fruits = ["apple", "banana", "cherry", "kiwi", "mango"]  
  
newlist = [x for x in fruits if "a" in x]  
  
print(newlist)
```

The Syntax

```
newlist = [expression for item in iterable if condition == True]
```

The return value is a new list, leaving the old list unchanged.

Condition

The *condition* is like a filter that only accepts the items that valuate to `True`.

Example

Only accept items that are not "apple":

```
newlist = [x for x in fruits if x != "apple"]
```

Expression

The *expression* is the current item in the iteration, but it is also the outcome, which you can manipulate before it ends up like a list item in the new list:

Example

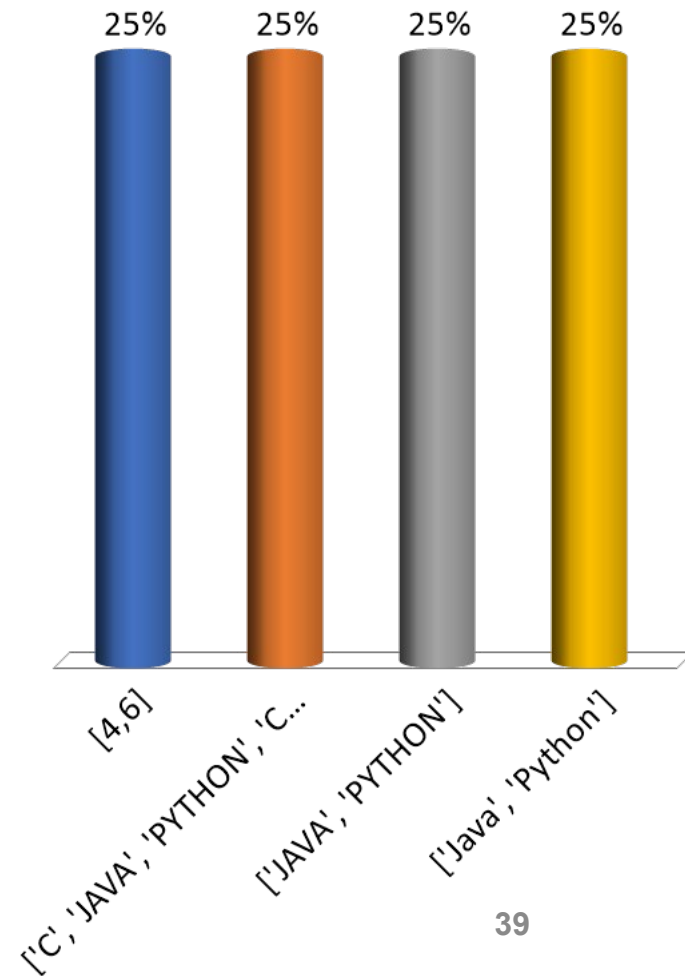
Set the values in the new list to upper case:

```
newlist = [x.upper() for x in fruits]
```

What is the output of the following code correct?

```
list1 = ["C", "Java", "Python", "C++"]  
print([ p.upper() for p in list1 if len(p)>=4])
```

- A. [4,6]
- B. ['C', 'JAVA', 'PYTHON', 'C++']
- ✓ C. ['JAVA', 'PYTHON']
- D. ['Java', 'Python']



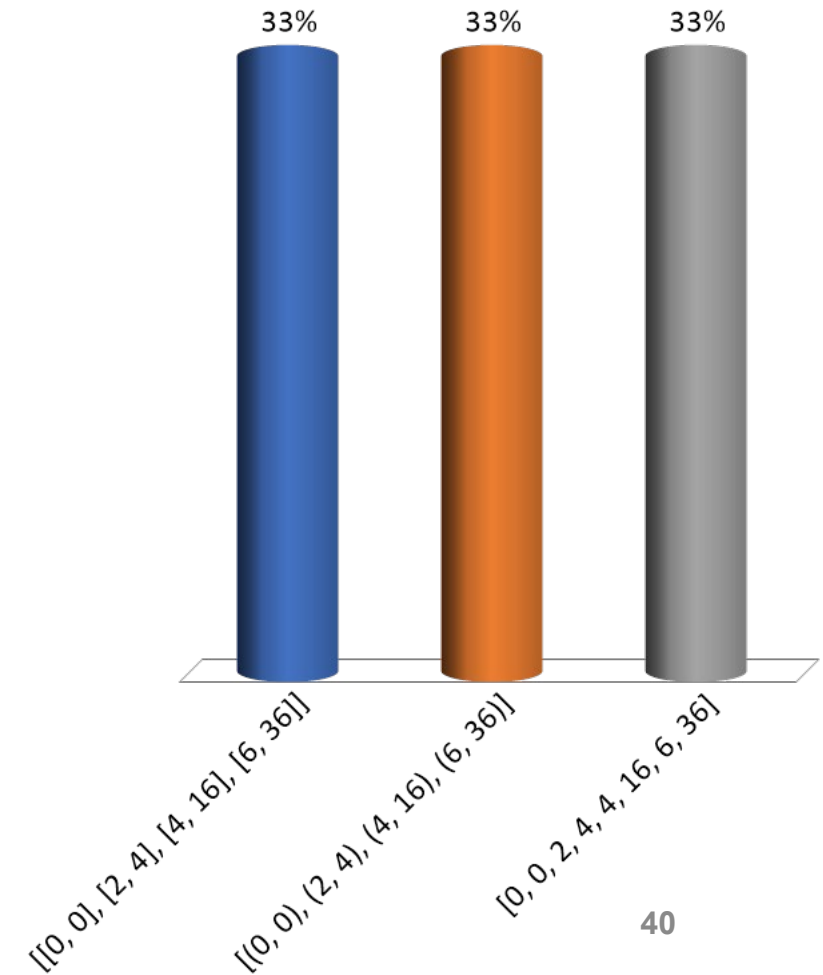
What is the output of the following code correct?

```
[(i,i**2) for i in range(8) if i%2 == 0]
```

A. `[[0, 0], [2, 4], [4, 16], [6, 36]]`

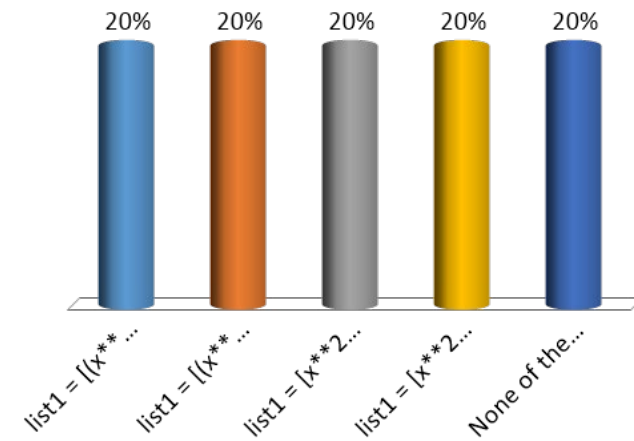
✓ B. `[(0, 0), (2, 4), (4, 16), (6, 36)]`

C. `[0, 0, 2, 4, 4, 16, 6, 36]`



Q5. Write a Python program, in the fewest number of lines possible, which creates a list of all the square numbers: x^2 (where $1 \leq x \leq 100$) that are divisible by 3. Which of the following statement is correct?

- A. `list1 = [(x**2)%3 for x in range(1,100) if x**2 == 0]`
- B. `list1 = [(x**2)%3 for x in range(1,101) if x**2 == 0]`
- C. `list1 = [x**2 for x in range(1,100) if x**2 % 3 == 0]`
- ✓ D. `list1 = [x**2 for x in range(1,101) if x**2 % 3 == 0]`
- E. None of the above



List comprehensions provide a concise way to create lists.

It consists of brackets containing an expression followed by a for clause, then zero or more for or if clauses. The expressions can be anything, meaning you can put in all kinds of objects in lists.

The result will be a new list resulting from evaluating the expression in the context of the for and if clauses which follow it.

The list comprehension always returns a result list.

```
new_list = [expression(i) for i in old_list]
```

```
new_list = [expression(i) for i in old_list if filter(i)]
```

```
new_list = [expression(i, j) for i in old_list1 for j in old_list2 if filter(i) if filter(j)]
```

Other Examples

```
[x + y for x in range(1,5) for y in range (1,4)]
```

It is as if we had done the following:

```
myList = [ ]  
for x in range (1,5):  
    for y in range (1,4):  
        myList.append(x+y)
```

[2, 3, 4, 3, 4, 5, 4, 5, 6, 5, 6, 7]