Dhananjay Porwal

EM19CS301110

Assignment 2

Ques1. Explain list, set, tuple, string?

Ans1. List:The simplest data structure in Python and is used to store a list of values. Lists are collections of items (strings, integers, or even other lists).

Each item in the list has an assigned index value.Lists are enclosed in [ ].

Each item in a list is separated by a comma. Unlike strings, lists are mutable, which means they can be changed.Lists hold a sequence of values (just like strings can hold a sequence of characters).

For example: list1 = ['one',2,3.4,'nishant']

Sets:Sets are unordered.Set elements are unique. Duplicate elements are not allowed.

A set itself may be modified, but the elements contained in the set must be of an immutable type. Sets can be used to perform mathematical set operations like union, intersection, symmetric difference etc.

For example: x = set(['foo', 'bar', 'baz', 'foo', 'qux'])

Tuples:A tuple is a sequence of values much like a list. Th

values stored in a tuple can be any type, and they are indexed by integers.The important difference is that tuples are immutable.That we can't change the elements of tuple once it is assigned whereas in the list, elements can be changed.
For example: t = (1, 'raju', 28, 'abc')

Strings:Strings in python are surrounded by either single quotation marks, or double quotation marks.'hello' is the same as "hello". You can display a string literal with the print() function.Like many other popular programming languages, strings in Python are arrays of bytes representing unicode characters.However, Python does not have a character data type, a single character is simply a string with a length of 1.Square brackets can be used to access elements of the string.
For example: a = "Hello, World!"

Ques2. Explain all method of above mentioned data types?

Ans2. List:

ethod Description
append() Adds an element at the end of the list
clear() Removes all the elements from the list
copy() Returns a copy of the list
count() Returns the number of elements with the specifie
d value
extend() Add the elements of a list (or any iterable), to
the end of the current list
index() Returns the index of the first element with the s
pecified value
insert() Adds an element at the specified position
pop() Removes the element at the specified position
remove() Removes the first item with the specified value
reverse() Reverses the order of the list
sort() Sorts the list

Set:
Method Description
add() Adds an element to the set
clear() Removes all the elements from the set

opy() Returns a copy of the set
difference() Returns a set containing the difference betw
een two or more sets
difference_update() Removes the items in this set that ar
e also included in another, specified set
discard() Remove the specified item
intersection() Returns a set, that is the intersection of
two other sets
intersection_update() Removes the items in this set that
are not present in other, specified set(s)
isdisjoint() Returns whether two sets have a intersection
or not
issubset() Returns whether another set contains this set
or not
issuperset() Returns whether this set contains another s
et or not
pop() Removes an element from the set
remove() Removes the specified element
Symmetric_difference() Returns a set with the symmetric
differences of two sets

ymmetric_difference_update() inserts the symmetric differences from this set and another
union() Return a set containing the union of sets
update() Update the set with the union of this set and others

Tuples:
Method Description
count() Returns the number of times a specified value occurs in a tuple
index() Searches the tuple for a specified value and returns the position of where it was found

Strings:
Method Description
capitalize() Converts the first character to upper case
casefold() Converts string into lower case
center() Returns a centered string
count() Returns the number of times a specified value occurs in a string

nCode() Returns an encoded version of the string
endswith() Returns true if the string ends with the spec
ified value
expandtabs() Sets the tab size of the string
find() Searches the string for a specified value and retu
rns the position of where it was found
format() Formats specified values in a string
format_map() Formats specified values in a string
index() Searches the string for a specified value and retu
rns the position of where it was found
isalnum() Returns True if all characters in the string a
re alphanumeric
isalpha() Returns True if all characters in the string ar
e in the alphabet
isdecimal() Returns True if all characters in the string a
re decimals
isdigit() Returns True if all characters in the string are
digits
isidentifier() Returns True if the string is an identifier
islower() Returns True if all characters in the string ar

lower case

isnumeric() Returns True if all characters in the string are numeric

isprintable() Returns True if all characters in the strin g are printable

isspace() Returns True if all characters in the string ar e whitespaces

istitle() Returns True if the string follows the rules of a title

isupper() Returns True if all characters in the string ar e upper case

join() Joins the elements of an iterable to the end of th e string

ljust() Returns a left justified version of the string

lower() Converts a string into lower case

lstrip() Returns a left trim version of the string

maketrans() Returns a translation table to be used in t ranslations

partition() Returns a tuple where the string is parted in to three parts

eplace() Returns a string where a specified value is repla
ced with a specified value
rfind() Searches the string for a specified value and retu
rns the last position of where it was found
rindex() Searches the string for a specified value and ret
urns the last position of where it was found
rjust() Returns a right justified version of the string
rpartition() Returns a tuple where the string is parted
into three parts
rsplit() Splits the string at the specified separator, and r
eturns a list
rstrip() Returns a right trim version of the string
split() Splits the string at the specified separator, and re
turns a list
splitlines() Splits the string at line breaks and returns a
list
startswith() Returns true if the string starts with the
specified value
strip() Returns a trimmed version of the string
swapcase() Swaps cases, lower case becomes upper case an

vice versa
title() Converts the first character of each word to uppe
r case
translate() Returns a translated string
upper() Converts a string into upper case
zfill() Fills the string with a specified number of 0 value
s at the beginning

Ques3. Explain indexing and slicing in each?

Ans3. Indexing" means referring to an element of an iterable b
y its position within the iterable. "Slicing" means getting a
subset of elements from an iterable based on their indice
s.

For example:

```
my_list = [_ for _ in 'abcdefghi']
my_list
['a', 'b', 'c', 'd', 'e', 'f', 'g', 'h', 'i']
```

Indexing

To retrieve an element of the list, we use the index oper

or ([]):
my_list[0]
'a'

Lists are "zero indexed", so [0] returns the zero-th (i.e. th e left-most) item in the list, and [1] returns the one-th i tem (i.e. one item to the right of the zero-th item). Since t here are 9 elements in our list ([0] through [8]), attempt ing to access my_list[9] throws an IndexError: list index out of range, since it is actually trying to get the tenth element, and there isn't one.
Python also allows you to index from the end of the list using a negative number, where [-1] returns the last ele ment. This is super-useful since it means you don't have t o programmatically find out the length of the iterable in order to work with elements at the end of it. The indice s and reverse indices of my_list are as follows:

['a', 'b', 'c', 'd', 'e', 'f', 'g', 'h', 'i']

↑ ↑ ↑ ↑ ↑ ↑ ↑ ↑ ↑

-9 -8 -7 -6 -5 -4 -3 -2 -1

Slicing

A slice is a subset of list elements. In the case of lists, a single slice will always be of contiguous elements. Slice notation takes the form

my_list[start:stop:stepover]

where start is the index of the first element to include, and stop is the index of the item to stop at without including it in the slice. So my_list[1:5] returns ['b', 'c', 'd', 'e']:

0 1 2 3 4 5 6 7 8

x ↓ ↓ ↓ ↓ x x x x

['a', 'b', 'c', 'd', 'e', 'f', 'g', 'h', 'i']

Ques4. Explain str(), globals(), locals(), vars(), eval(), exec(), execfile(), repr(), ascii().

Ans4. str()

The str() function returns the string version of the give

object.

The syntax of str() is:

str(object, encoding='utf-8', errors='strict')

str() Parameters

The str()method takes three parameters:

object - The object whose string representation is to be returned. If not provided, returns the empty string

encoding - Encoding of the given object. Defaults of UTF-8 when not provided.

errors - Response when decoding fails. Defaults to 'strict'. There are six types of errors:

strict - default response which raises a UnicodeDecodeError exception on failure

ignore - ignores the unencodable Unicode from the result

replace - replaces the unencodable Unicode to a question mark

xmlcharrefreplace - inserts XML character reference instead of unencodable Unicode

backslashreplace - inserts a \uNNNN espace sequence instead of unencodable Unicode

amereplace - inserts a \N{...} escape sequence instead of u
nencodable Unicode
Return value from str()
The str() method returns a string, which is considered an
informal or nicely printable representation of the given
object.

globals()
Return a dictionary representing the current global symb
ol table. This is always the dictionary of the current mod
ule (inside a function or method, this is the module where
it is defined, not the module from which it is called).

locals()
Update and return a dictionary representing the curren
t local symbol table. Free variables are returned by loca
ls() when it is called in function blocks, but not in class
blocks. Note that at the module level, locals() and globals(
) are the same dictionary.

ars()

Return the __dict__ attribute for a module, class, instance, or any other object with a __dict__ attribute.Objects such as modules and instances have an updateable __dict__ attribute; however, other objects may have write restrictions on their __dict__ attributes (for example, classes use a types.MappingProxyType to prevent direct dictionary updates).

Without an argument, vars() acts like locals(). Note, the locals dictionary is only useful for reads since updates to the locals dictionary are ignored.A TypeError exception is raised if an object is specified but it doesn't have a __dict__ attribute (for example, if its class defines the __slots__ attribute).

eval()

The arguments are a string and optional globals and locals. If provided, globals must be a dictionary. If provided, locals can be any mapping object.The expression argument is parsed and evaluated as a Python expression (technicall

speaking, a condition list) using the globals and locals dictionaries as global and local namespace. If the globals dictionary is present and does not contain a value for the key __builtins__, a reference to the dictionary of the built-in module builtins is inserted under that key before expression is parsed. This means that expression normally has full access to the standard builtins module and restricted environments are propagated. If the locals dictionary is omitted it defaults to the globals dictionary. If both dictionaries are omitted, the expression is executed with the globals and locals in the environment where eval() is called. Note, eval() does not have access to the nested scopes (non-locals) in the enclosing environment. The return value is the result of the evaluated expression. Syntax errors are reported as exceptions. Example:

```
>>>
x = 1
eval('x+1')
2
```

This function can also be used to execute arbitrary code objects (such as those created by compile()). In this case pa ss a code object instead of a string. If the code object has been compiled with 'exec' as the mode argument, eval()'s re turn value will be None.
Hints: dynamic execution of statements is supported by th e exec() function. The globals() and locals() functions ret urns the current global and local dictionary, respectively , which may be useful to pass around for use by eval() or exec().See ast.literal_eval() for a function that can saf ely evaluate strings with expressions containing only lit erals.Raises an auditing event exec with the code object a s the argument. Code compilation events may also be raise d.

exec()
This function supports dynamic execution of Python code. object must be either a string or a code object. If it is a s tring, the string is parsed as a suite of Python statemen ts which is then executed (unless a syntax error occurs)

1 If it is a code object, it is simply executed. In all cases, the code that's executed is expected to be valid as file inp ut (see the section "File input" in the Reference Manual) . Be aware that the nonlocal, yield, and return statemen ts may not be used outside of function definitions even w ithin the context of code passed to the exec() function. Th e return value is None.In all cases, if the optional parts are omitted, the code is executed in the current scope. If only globals is provided, it must be a dictionary (and not a subclass of dictionary), which will be used for both the global and the local variables. If globals and locals are g iven, they are used for the global and local variables, re spectively. If provided, locals can be any mapping object. Re member that at module level, globals and locals are the sa me dictionary. If exec gets two separate objects as globals and locals, the code will be executed as if it were embed ded in a class definition. If the globals dictionary does no t contain a value for the key __builtins__, a reference to the dictionary of the built-in module builtins is insert ed under that key. That way you can control what built

s are available to the executed code by inserting your own __builtins__ dictionary into globals before passing it to exec(). Raises an auditing event exec with the code object as the argument. Code compilation events may also be raised.

execfile ()
A file to be parsed and evaluated as a sequence of Python statements (similarly to a module).Any mapping object providing global namespace. Any mapping object providing local namespace.

repr(object)
Return a string containing a printable representation of an object. For many types, this function makes an attempt to return a string that would yield an object with the same value when passed to eval(), otherwise the representation is a string enclosed in angle brackets that contains the name of the type of the object together with additional information often including the name and address o

the object. A class can control what this function returns for its instances by defining a __repr__() method.

ascii(object)
As repr(), return a string containing a printable representation of an object, but escape the non-ASCII characters in the string returned by repr() using \x, \u or \U escapes. This generates a string similar to that returned by repr() in Python 2.

Ques5. Explain looping and control flow with program?

Ans5. Python programming language provides following types of loops to handle looping requirements.

While Loop

Syntax :
```
while expression:
    statement(s)
```
In Python, all the statements indented by the same numbe

of character spaces after a programming construct are co nsidered to be part of a single block of code. Python uses i ndentation as its method of grouping statements.

```
# prints Hello Geek 3 Times
count = 0
while (count < 3):
    count = count+1
    print("Hello Geek")
```

Output:

```
Hello Geek
Hello Geek
Hello Geek
```

See this for an example where while loop is used for iter ators. As mentioned in the article, it is not recommended to use while loop for iterators in python.

For in Loop

In Python, there is no C style for loop, i.e., for (i=0; i<n; i++). There is "for in" loop which is similar to for each loop in other languages.
Syntax:

```
for iterator_var in sequence:
    statements(s)
```

It can be used to iterate over iterators and a range.

```
# Iterating over a list
print("List Iteration")
l = ["geeks", "for", "geeks"]
for i in l:
    print(i)
```

```
 Iterating over a tuple (immutable)
print("\nTuple Iteration")
t = ("geeks", "for", "geeks")
for i in t:
    print(i)

# Iterating over a String
print("\nString Iteration")
s = "Geeks"
for i in s :
    print(i)

# Iterating over dictionary
print("\nDictionary Iteration")
d = dict()
d['xyz'] = 123
d['abc'] = 345
for i in d :
    print("%s  %d" %(i, d[i]))
```

Output:

List Iteration

geeks

for

geeks

Tuple Iteration

geeks

for

geeks

String Iteration

G

e

e

k

s

Dictionary Iteration

xyz 123

abc 345

e can use for in loop for user defined iterators. See this for example.

Nested Loops

Python programming language allows to use one loop inside another loop. Following section shows few examples to illustrate the concept.

Syntax:

```
for iterator_var in sequence:
    for iterator_var in sequence:
        statements(s)
        statements(s)
```

The syntax for a nested while loop statement in Python programming language is as follows:

```
while expression:
```

```
while expression:
    statement(s)
    statement(s)
```

A final note on loop nesting is that we can put any type of loop inside of any other type of loop. For example a for loop can be inside a while loop or vice versa.

```
from __future__ import print_function
for i in range(1, 5):
    for j in range(i):
        print(i, end=' ')
    print()
```

Output:

```
1
2 2
3 3 3
4 4 4 4
```

Loop Control Statements

Loop control statements change execution from its normal sequence. When execution leaves a scope, all automatic objects that were created in that scope are destroyed. Python supports the following control statements.

Continue Statement

It returns the control to the beginning of the loop.

```
# Prints all letters except 'e' and 's'
for letter in 'geeksforgeeks':
    if letter == 'e' or letter == 's':
        continue
    print 'Current Letter :', letter
    var = 10
```

Output:

```
Current Letter : g
Current Letter : k
Current Letter : f
Current Letter : o
```

urrent Letter : r
Current Letter : g
Current Letter : k
Break Statement
It brings control out of the loop

```
for letter in 'geeksforgeeks':

    # break the loop as soon it sees 'e'
    # or 's'
    if letter == 'e' or letter == 's':
        break

print 'Current Letter :', letter
```

Output:

Current Letter : e
Pass Statement
We use pass statement to write empty loops. Pass is also

ed for empty control statement, function and classes.

```
# An empty loop
for letter in 'geeksforgeeks':
    pass
print 'Last Letter :', letter
```

Output:

```
Last Letter : s
```