IQRA NATIONAL UNIVERSITY

INU

DEFINING POSSIBILITIES, CREATING OPPORTUNITIES

Name: Inzamam
ID#12998
Subject: Data Sciences
Program: BS(SE)

Question No. 1:

(b):

Ans: A variable can have a short name such as x and y or more like (age, carname, total_volume).

Rules:

. A variable name cannot start with a number

.A variable name must start with a letter or the underscore character

. Variable names are case-sensitive (age, Age and AGE are three different variables)

. A variable name can only contain alpha-numeric characters and underscores (A-z, 0-9, and _).

Example:

```
#Legal variable names:
myvar = "John"
my_var = "John"
_my_var = "John"
myVar = "John"
MYVAR = "John"
myvar2 = "John"
#Illegal variable names:
2myvar = "John"
my-var = "John"
my var = "John"
```

(a):

Ans:

Variable in Python:

A Python variable is a reserved memory location to store values. Actually a variable in a python program gives information to the computer for processing.

In Python every single value has a datatype. There are various data types in Python such as Numbers, List, Tuple, Strings, Dictionary, etc. Variables can be declared by any name or even alphabets like a, aa, abc, etc.

<u>How to Declare and use a Variable:</u>

lets pick up an example. We will declare variable "a" and print it.

<u>a=100</u>
<u>print (a)</u>

<u>Re-declare a Variable:</u>

We can also re-declare a variable even when we declared it before.
Here we have variable initialized to f=0.
Later, we re-assign the variable f to value ‘Inzu123’

# Declare a variable and initialize it
f = 0
print f
# re-declaring the variable works
f = 'Inzu123'
print f

<u>Concatenate Variables:</u>

For example, we will concatenate "Inzu" with the number "123".
Not like Java, which concatenates number with string without declaring number as string, Python requires declaring the number as string or else it will show a TypeError,

#ERROR: different types cannot be combined
print(“Inzu”+123)

means we will get undefined output like,

a="Inzu"
b = 123
print a+b

<u>Local & Global Variables:</u>

1. Variable "f" is global in scope and is assigned value 101 which is printed in output

2.Variable f is again declared in function and assumes local scope. It is assigned value "I am learning Python." which is printed out as an output. This variable is different from the global variable "f" define earlier
3. Once the function call is over, the local variable f is destroyed. At line 12, when we again, print the value of "f" is it displays the value of global variable f=101.

Lets explain it in an example,

```
# Declare a variable and initialize it
f = 101
print f
# Global vs. local variables in functions
def someFunction():
# global f
   f = 'I am learning Python'
   print f
someFunction()
print f
```

Delete a variable:
You can also delete variable by using the command del "variable name".

In the example below, we deleted variable f, and when we proceed to print it, we get error "variable name is not defined" which means you have deleted the variable.

```
#Declare a variable and initialize it
f = 11;
print(f)

del f
print(f)
```

Hence the variable is deleted now.

<u>Question No. 2:</u>

<u>(b):</u>

<u>Ans:</u>

The str() method allows you to convert an integer to a string in Python. The syntax for this method is:

str(number_to_convert)

lets pick an example to indicate how this method actually works.

raw_user_age = input("What is your age?")
user_age = int(raw_user_age)

we wish to print a message with our user's age to the console. We could try this by using the this code:

print("Your age is: " + user_age)

but this code give us an error:

Traceback (most recent call last):

```
File "main.py", line 3, in <module>
        print("Your age is: " + user_age)
TypeError: can only concatenate str (not "int") to str
```

it is because we can not concatenate a string to an integer like we done above before. To make this code work, we have to convert user's age to a string. We are able to try this by using the str() method:

```python
raw_user_age = input("What is your age?")
user_age = int(raw_user_age)
as_string = str(user_age)

print("Your age is: " + as_string)
```

This code returns this:

```
What is your age?

13

Your age is: 13
```

We have successfully converted our integer to a string using str(). Both values are now strings, which means that we can now concatenate the message **Your age is:** with the user's age.

(a):

Ans:

Data Types:

In programming, a data type is a classification that specifies what kind of value a variable has and what type of mathematical, relational or logical operations can be applied to it without causing an error. For example, a string is a data type that is used to classify text and an integer is a data type used to classify whole numbers.
The data type defines which operations can safely be performed to create, transform and use the variable in another computation. When a program language requires a

variable to only be used in ways that respect its data type, that language is said to be strongly typed. This could not cause errors, because while it is logical to ask the computer to multiply a float by an integer (1.4 x 7), it is illogical to ask the computer to multiply a float by a string (1.4 x Alice). When a programming language allows a variable of one data type to be used as if it were a value of another data type, the language is called weakly typed.

<u>Data types that are used in Python:</u>

<u>Numbers:</u>
Python numbers variables are created by the standard Python method:

```
var = 382
```

Most of the time using the standard Python number type is fine. Python will automatically convert a number from one type to another if it requires. But, sometimes a specific number type is needed (ie. complex, hexidecimal), the format can be forced into a format by using additional syntax in the table below:

Type			**Format**			**Description**
int			a = 10			Signed Integer
long			a = 345L			(L) Long integers, they can also be represented in octal and hexadecimal
float			a = 45.67			(.) Floating point real values
complex			a = 3.14J			(J) Contains integer in the range 0 to 255.

Most of the time Python will do variable conversion automatically. You can also use Python conversion functions (int(), long(), float(), complex()) to convert data from one type to another. For example:

```
message = "Good morning"
num = 85
pi = 3.14159

print(type(message))  # This will return a string
print(type(n))  # This will return an integer
print(type(pi))  # This will return a float
```

String:
Create string variables by enclosing characters in quotes. Python uses single quotes ' double quotes " and triple quotes """ to denote literal strings. Only the triple quoted strings """. Such as,

```
firstName = 'john'
lastName = "smith"
message = """This is a string that will span across multiple lines. Using newline characters and no spaces for the next lines. The end of lines within this string also count as a newline when printed"""
```

Strings can be accessed as a whole string, for example,

```
var1 = 'Hi Inzu!'
var2 = 'RhinoPython'

print var1[0] # this will print the first character in the string an `H`
print var2[1:5] # this will print the substring 'hinoP`
```

List:
Lists are a very useful variable type in Python. A list can contain a series of values. List variables are declared by using brackets [ ] following the variable name.

```
A = [ ] # This is a blank list variable
B = [1, 23, 45, 67] # this list creates an initial list of 4 numbers.
C = [2, 4, 'john'] # lists can contain different variable types.
```

You can assign data to a specific element of the list using an index into the list. The list index starts at zero. Data can be assigned to the elements of an array as follows:

```
mylist = [0, 1, 2, 3]
mylist[0] = 'Rhino'
mylist[1] = 'Grasshopper'
mylist[2] = 'Flamingo'
```

```
mylist[3] = 'Bongo'
print mylist[1]
```

<u>Tuple:</u>

Tuples are a group of values like a list and are manipulated in similar ways. But, tuples are fixed in size once they are assigned. In Python the fixed size is considered immutable as compared to a list that is dynamic and mutable. Tuples are defined by parenthesis ().

```
myGroup = ('Rhino', 'Grasshopper', 'Flamingo', 'Bongo')
```

Here are some advantages of tuples over lists:

1. Elements to a tuple. Tuples have no append or extend method.
2. Elements cannot be removed from a tuple.
3. You can find elements in a tuple, since this doesn’t change the tuple.
4. You can also use the in operator to check if an element exists in the tuple.
5. Tuples are faster than lists. If you’re defining a constant set of values and all you’re ever going to do with it is iterate through it, use a tuple instead of a list.

<u>Dictionary:</u>

Dictionaries in Python are lists of Key:Value pairs. This is a very powerful datatype to hold a lot of related information that can be associated through keys. The main operation of a dictionary is to extract a value based on the key name.

Dictionaries are created by using braces ({}) with pairs separated by a comma (,) and the key values associated with a colon(:). In Dictionaries the Key must be unique. Here is a quick example on how dictionaries might be used:

```
room_num = {'john': 425, 'tom': 212}

room_num['john'] = 645  # set the value associated with the 'john' key to 645
```

print (room_num['tom']) # print the value of the 'tom' key.

room_num['isaac'] = 345 # Add a new key 'isaac' with the associated value

print (room_num.keys()) # print out a list of keys in the dictionary

print ('isaac' in room_num) # test to see if 'issac' is in the dictionary. This returns true.

Dictionaries are very complex to understand but easy to store data that can be easily to be accessed.

Question No. 3:
Ans:

Python print()

The print() function is used to print the given object to an output device (screen) or to the text stream file.

The full syntax of print() is:

print(*objects, sep=' ', end='\n', file=sys.stdout, flush=False)

print() Parameters

- **objects** - object to the printed. ***** indicates that there may be more than one object
- **sep** - objects are separated by sep. (Default value: ' ')
- **end** - end is printed at last
- **file** - must be an object with write(string) method. If omitted it, sys.stdout will be used which prints objects on the screen.
- **flush** - If True, the stream is forcibly flushed. (Default value: False)

How print() works in Python?

print("Iqra National University.")

a = 5

```
# Two objects are passed

print("a =", a)


b = a

# Three objects are passed

print('a =', a, '= b')
```

Output

```
Iqra National University.

a = 5

a = 5 = b
```

print() with separator and end parameters

```
a = 5

print("a =", a, sep='00000', end='\n\n\n')

print("a =", a, sep='0', end='')
```

Output

```
a =000005

a =05
```

print() with file parameter

```
sourceFile = open('python.txt', 'w')

print(' Iqra National University ', file = sourceFile)

sourceFile.close()
```

This program tries to open the python.txt in writing mode. If this file doesn't exist, python.txt file is created and opened in writing mode.

Python type()

The type() function either returns the type of the object or returns a new type object based on the arguments passed.

The type() function has two different forms:

```
type(object)

type(name, bases, dict)
```

type() With a Single Object Parameter

If a single object is passed to type(), the function returns its type.

Get Type of an Object

```
numbers_list = [1, 2]

print(type(numbers_list))


numbers_dict = {1: 'one', 2: 'two'}

print(type(numbers_dict))


class Foo:

    a = 0


foo = Foo()

print(type(foo))
```

Output

```
<class 'dict'>

<class 'Foo'>

<class '__main__.Foo'>
```

type() With name, bases and dict Parameters

If three parameters are passed to type(), it returns a new **type** object.

The three parameters are:

name	a class name; becomes the __name__ attribute
bases	a tuple that itemizes the base class; becomes the __bases__ attribute
dict	a dictionary which is the namespace containing definitions for the class body; becomes the __dict__ attribute

Create a type object

```
o1 = type('X', (object,), dict(a='Foo', b=12))


print(type(o1))

print(vars(o1))


class test:

  a = 'Foo'

  b = 12


o2 = type('Y', (test,), dict(a='Foo', b=12))

print(type(o2))

print(vars(o2))
```

Output

```
<class 'type'>

{'b': 12, 'a': 'Foo', '__dict__': <attribute '__dict__' of 'X' objects>, '__doc__': None, '__weakref__': <attribute '__weakref__' of 'X' objects>}

<class 'type'>

{'b': 12, 'a': 'Foo', '__doc__': None}
```

Real-Life Usage of type() function

Python is a dynamically-typed language. So, if we want to know the type of the arguments, we can use the type() function. If you want to make sure that your function works only on the specific types of objects, use isinstance() function.

Let’s say we want to create a function to calculate something on two integers. We can implement it in the following way.

```
def calculate(x, y, op='sum'):
    if not(isinstance(x, int) and isinstance(y, int)):
        print(f'Invalid Types of Arguments - x:{type(x)}, y:{type(y)}')
        raise TypeError('Incompatible types of arguments, must be integers')


    if op == 'difference':
        return x - y
    if op == 'multiply':
        return x * y
    # default is sum
    return x + y
```

The isinstance() function is used to validate the input argument type. The type() function is used to print the type of the parameters when validation fails.

Question No. 4:

Ans:

One of the most common forms of reassignment is an **update** where the new value of the variable depends on the old. For example,

x = x + 1

This means get the current value of x, add one, and then update x with the new value. The new value of x is the old value of x plus 1. Although this assignment statement may look a bit strange, remember that executing assignment is a two-step process. First, evaluate the right-hand side expression. Second, let the variable name on the left-hand side refer to this new resulting object. The fact that x appears on both sides does not matter. The semantics of the assignment statement makes sure that there is no confusion as to the result. The visualizer makes this very clear.

```
x = 6
x = x + 1
```

```
x = 6      # initialize x

print(x)

x = x + 1    # update x

print(x)
```

If you try to update a variable that doesn’t exist, you get an error because Python evaluates the expression on the right side of the assignment operator before it assigns the resulting value to the name on the left. Before you can update a variable, you have to initialize it, usually with a simple assignment. In the above example, x was initialized to 6.
Updating a variable by adding something to it is called an increment; subtracting is called a decrement. Sometimes programmers talk about incrementing or decrementing without specifying by how much; when they do they usually mean by 1. Sometimes programmers also talk about bumping a variable, which means the same as incrementing it by 1.
Incrementing and decrementing are such common operations that programming languages often include special syntax for it. In Python += is used for incrementing, and -= for decrementing. In some other languages, there is even a special syntax ++ and -- for incrementing or decrementing by 1. Python does not have such a special syntax. To increment x by 1 you have to write x += 1 or x = x + 1.

```
x = 6       # initialize x

print(x)

x += 3      # increment x by 3; same as x = x + 3

print(x)

x -= 1      # decrement x by 1

print(x)
```

Imagine that we wanted to not increment by one each time but instead add together the numbers one through ten, but only one at a time.

```
s = 1

print(s)

s = s + 2

print(s)

s = s + 3
```

```
print(s)

s = s + 4

print(s)

s = s + 5

print(s)

s = s + 6

print(s)

s = s + 7

print(s)

s = s + 8

print(s)

s = s + 9

print(s)

s = s + 10

print(s)
```

After the initial statement, where we assign s to 1, we can add the current value of s and the next number that we want to add (2 all the way up to 10) and then finally reassign that that value to s so that the variable is updated after each line in the code.

This will be tedious when we have many things to add together. Later you will read about an easier way to do this kind of task.

Question No. 5:
Ans:

Errors:

Errors or mistakes in a program are often referred to as bugs. They are almost always the fault of the programmer. The process of finding and eliminating errors is called debugging. Errors can be classified into three major groups:

- Syntax errors
- Runtime errors
- Logical errors

Syntax errors:

Python can find these sorts of errors once it tries to dissect your program, and exit with an error message while not running anything. Syntax errors are mistakes within the use of the Python language, and are analogous to spelling or synchronic linguistics mistakes in a language like English: for instance, the sentence Would you some tea? doesn't make sense – it's missing a verb.

Common Python syntax errors include:

- leaving out a keyword
- putting a keyword in the wrong place
- leaving out a symbol, like a colon, comma or brackets
- misspelling a keyword

- Incorrect indentation
- Empty block

Python can do its best to inform you wherever the error is found, however generally its messages are often misleading: for instance, if you forget to escape a quotation mark within a string you will get a syntax error referring to a place later in your code, even supposing that's not the real source of the problem. If you can’t see anything wrong on the line laid out in the error message, attempt backtracking through the last few lines. As you program more, you'll improve at distinctive and fixing errors.

Examples of syntax errors in Python are as follows:

```
myfunction(x, y):
return x + y
else:
print("Hello!")
if mark >= 50
print("You passed!")

if arriving:
print("Hi!")
esle:
print("Bye!")

if flag:
print("Flag is set!")
```

<u>Runtime errors:</u>

If a program is syntactically correct – that's, free of syntax errors – it'll be run by the Python interpreter. However, the program could exit unexpectedly throughout execution if it encounters a run-time error – a problem that wasn't detected when the program was parsed, however is simply unconcealed once a specific line is executed. Once a program comes to a halt due to a run-time error, we say that it's crashed.

Consider the English instruction flap your arms and fly to Australia. Whereas the instruction is structurally correct and you'll perceive its meaning perfectly, it's not possible for you to follow it.

Examples of Python runtime errors are as follows:

- Division by zero
- performing an operation on incompatible varieties
- using an identifier that has not been outlined
- accessing an inventory element, dictionary value or object attribute that doesn't exist
- trying to access a file that doesn't exist

Runtime errors typically sneak in if you don't think about all attainable values that a variable may contain, particularly when you are processing user input. You ought to continuously attempt to add checks to your code to create positive that it will manage bad input and edge cases graciously**.** We can solve these types of error by using exception handlings. In exception handling we can tell python what to do when an error occurs instead if crashing the app

<u>Logical errors:</u>

Logical errors are the foremost tough to fix. They occur once the program runs while not crashing, however produces an incorrect result. The error is caused by a mistake within the program's logic. You won't get an error message, because no syntax or run-time error has occurred. You'll have to be compelled to realize the matter on your own by reviewing all the relevant elements of your code – though some tools will flag suspicious code that seems like it may cause surprising behavior.

Sometimes there are often absolutely nothing wrong along with your Python implementation of an algorithm – the algorithm itself are often incorrect. However, more often these sorts of errors are caused by coder carelessness. Here are some examples of mistakes that result in logical errors:

- using the incorrect variable name
- indenting a block to the incorrect level
- using integer division rather than floating-point division
- getting operator precedence wrong
- making an error in a mathematician expression

- Off-by-one, and different numerical errors

If you misspell an identifier name, you will get a run-time error or a logical error, depending on whether or not the misspelled name is outlined.

A common source of variable name mix-ups and incorrect indentation is frequent repetition and pasting of enormous blocks of code. If you've got several duplicate lines with minor variations, it’s terribly straightforward to miss a necessary change after you are editing your pasted lines. You ought to continuously try and factor out excessive duplication using functions and loops.