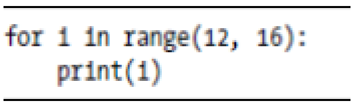
APPLICATION DEVELOPMENT USING PYTHON

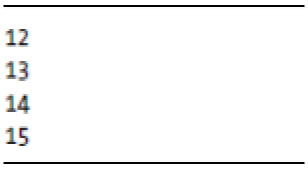
INTERNAL ASSESSMENT 1

MARKING SCHEME

1. A) If we want to execute a block of code only a certain number of times then we can do this with a for loop statement and the range() function.
Some functions can be called with multiple arguments separated by a comma, and range() is one of them.
This lets us change the integer passed to range() to follow any sequence of integers, including starting at a number other than zero.
v

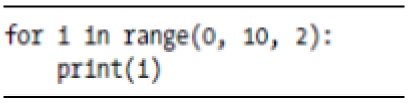


```
for i in range(12, 16):
    print(i)
```

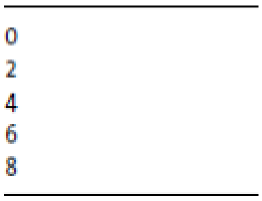


```
12
13
14
15
```

The range() function can also be called with three arguments. The first two arguments will be the start and stop values, and the third will be the step argument. The step is the amount that the variable is increased by after each iteration.

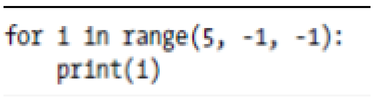


```
for i in range(0, 10, 2):
    print(i)
```

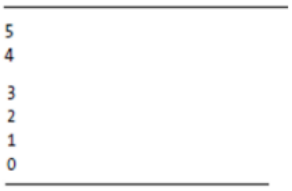


```
0
2
4
6
8
```

So calling range(0, 10, 2) will count from zero to eight by intervals of two.
The range() function is flexible in the sequence of numbers it produces for for loops. We can even use a negative number for the step argument to make the for loop count down instead of up.



```
for i in range(5, -1, -1):
    print(i)
```



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

Running a for loop to print i with range(5, -1, -1) should print from five down to zero.

```
//program
while(True):
   name=input("Enter name")
   if(name=="Joe"):
        password=input("Enter password")
        if(password=="swordfish"):
                  break
```

1.b) i) -10%4=2

```
4 ⌈ -10⌉ -3
 (-)-12
 ________
      2
```

ii) 2**3**2=512

(2)^((3)^2)

2^9

512

iii) not(False)

same as not(0)=1

True

2.a) i) None

In Python there is a value called None, which represents the absence of a value. None is the only value of the NoneType data type. This value-without-a-value can be helpful when we need to store something that won’t be confused for a real value in a variable.
One place where None is used is as the return value of print().
The print() function displays text on the screen, but it doesn’t need to return anything in the same way len() or input() does. But since all function calls need to evaluate to a return value, print() returns None.
Behind the scenes, Python adds return None to the end of any function definition with no return statement.

```
>>> spam = print('Hello!')
Hello!
>>> None == spam
True
```

ii) keyword arguments
keyword arguments are identified by the keyword put before them in the function call.
Keyword arguments are often used for optional parameters.
For example, the print() function hasthe optional parameters end and sep to specify what should be printed at the end of its arguments and between its arguments (separating them), respectively.
The two strings appear on separate lines because the print() function automatically adds a newline character to the end of the string it is passed. However, we can set the end keyword argument to change this to a different string. For example, if the program were this:

```
print('Hello')
print('World')
```

```
Hello
World
```

```
>>> print('cats', 'dogs', 'mice', sep=',')
cats,dogs,mice
```

```
print('Hello', end='')
print('World')
```

```
HelloWorld
```

The output is printed on a single line because there is no longer a new-line printed after 'Hello'. Instead, the blank string is printed. This is useful if we need to disable the newline that gets added to the end of every print() function call.
Similarly, when we pass multiple string values to print(), the function will automatically separate them with a single space. But we could replace the default separating string by passing the sep keyword argument.
iii) replication operator
The * operator is used for multiplication when it operates on two integer or floating-point values. But, when the * operator is used on one string value and one integer value, it becomes the string replication operator.

```
>>> 'Alice' * 5
'AliceAliceAliceAliceAlice'
```

iv) sys.exit()

we can cause the program to terminate, or exit, by calling the sys.exit() function. Since this function is in the sys module, we have to import sys before your program can use it. This program has an infinite loop with no break statement inside. The only way this program will end is if the user enters exit, causing sys.exit() to be called. When response is equal to exit, the program ends.

```
import sys

while True:
    print('Type exit to exit.')
    response = input()
    if response == 'exit':
        sys.exit()
    print('You typed ' + response + '.')
```

2.b)
```
import random
 import string
characters = string.ascii_letters + string.digits + string.punctuation
password = ''.join(random.choice(characters) for i in range(8))
print("Random password is:", password)
```

3.a) break Statements: There is a shortcut to getting the program execution to break out of a while loop's clause early. If the execution reaches a break statement, it immediately exits the while loop's clause. In code, a break statement simply contains the break keyword.

```
❶ while True:
       print('Please type your name.')
❷      name = input()
❸      if name == 'your name':
❹          break
❺ print('Thank you!')
```

Continue statements: Like break statements, continue statements are used inside loops. When the program execution reaches a continue statement, the program execution immediately jumps back to the start of the loop and reevaluates the loop's condition.

```
  while True:
      print('Who are you?')
      name = input()
❶     if name != 'Joe':
❷         continue
      print('Hello, Joe. What is the password? (It is a fish.)')
❸     password = input()
      if password == 'swordfish':
❹         break
❺ print('Access granted.')
```

3.b) If we don't want to crash the program due to errors instead we want the program to detect errors, handle them, and then continue to run. For example,

A ZeroDivisionError happens whenever we try to divide a number by zero. From the line number given in the error message, we know that the return statement in spam() is causing an error.

Errors can be handled with try and except statements. The code that could potentially have an error is put in a try clause. The program execution moves to the start of a following except clause if an error happens. We can put the previous divide-by-zero code in a try clause and have an except clause contain code to handle what happens when this error occurs.

```
def spam(divideBy):
    return 42 / divideBy

print(spam(2))
print(spam(12))
print(spam(0))
print(spam(1))
```

```
21.0
3.5
Traceback (most recent call last):
  File "C:/zeroDivide.py", line 6, in <module>
    print(spam(0))
  File "C:/zeroDivide.py", line 2, in spam
    return 42 / divideBy
ZeroDivisionError: division by zero
```

```
def spam(divideBy):
    try:
        return 42 / divideBy
    except ZeroDivisionError:
        print('Error: Invalid argument.')

print(spam(2))
print(spam(12))
print(spam(0))
print(spam(1))
```

```
21.0
3.5
Error: Invalid argument.
None
42.0
```

4.a) Parameters and variables that are assigned in a called function are said to exist in that function's local scope. Variables that are assigned outside all functions are said to exist in the global scope. A variable that exists in a local scope is called a local variable, while a variable that exists in the global scope is called a global variable. A variable must be one or the other; it cannot be both local and global. When a scope is destroyed, all the values stored in the scope's variables are forgotten. There is only one global scope, and it is created when your program begins. When your program terminates, the global scope is destroyed, and all its variables are forgotten.

A localscope is created whenever a function is called. Any variables assigned in this function exist within the local scope. When the function returns, the local scope is destroyed, and these variables are forgotten.

Local Variables Cannot Be Used in the Global Scope

```
def spam():
    eggs = 31337
spam()
print(eggs)
```

```
Traceback (most recent call last):
  File "C:/test3784.py", line 4, in <module>
    print(eggs)
NameError: name 'eggs' is not defined
```

Local Scopes Cannot Use Variables in Other Local Scopes

```
  def spam():
❶     eggs = 99
❷     bacon()
❸     print(eggs)

  def bacon():
      ham = 101
❹     eggs = 0

❺ spam()
```

Global Variables Can Be Read from a Local Scope

```
def spam():
    print(eggs)
eggs = 42
spam()
print(eggs)
```

Local and Global Variables with the Same Name

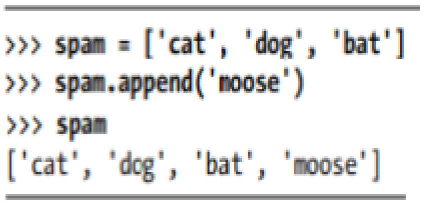
```
  def spam():
❶     eggs = 'spam local'
      print(eggs)    # prints 'spam local'

  def bacon():
❷     eggs = 'bacon local'
      print(eggs)    # prints 'bacon local'
      spam()
      print(eggs)    # prints 'bacon local'

❸ eggs = 'global'
  bacon()
  print(eggs)        # prints 'global'
```

```
bacon local
spam local
bacon local
global
```

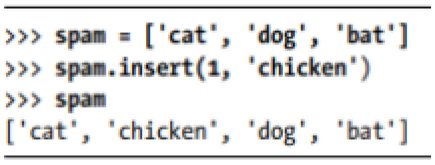
4.b)

```
def fact(n):
        if(n==0 or n==1):
                return 1
        else:
                return(n*fact(n-1))
num=int(input("Enter a number"))
print(fact(num))
```

5.a) A list is a value that contains multiple values in an ordered sequence.
To add new values to a list, use the append() and insert() methods.
The append() method call adds the argument to the end of the list.
The insert() method can insert a value at any index in the list. The first argument to insert() is the index for the new value, and the second argument is the new value to be inserted.
The append() and insert() methods are list methods and can be called only on list values, not on other values such as strings or integers.

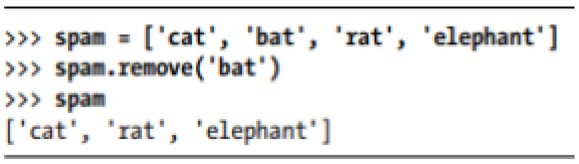


```
>>> spam = ['cat', 'dog', 'bat']
>>> spam.append('moose')
>>> spam
['cat', 'dog', 'bat', 'moose']
```



```
>>> spam = ['cat', 'dog', 'bat']
>>> spam.insert(1, 'chicken')
>>> spam
['cat', 'chicken', 'dog', 'bat']
```

The remove() method is passed the value to be removed from the list it is called on.

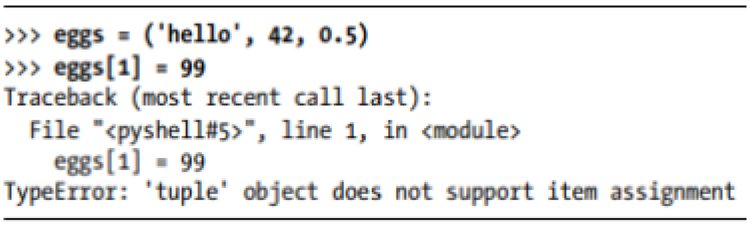


```
>>> spam = ['cat', 'bat', 'rat', 'elephant']
>>> spam.remove('bat')
>>> spam
['cat', 'rat', 'elephant']
```

5,b) An index will get a single value from a list, a slice can get several values from a list, in the form of a new list.

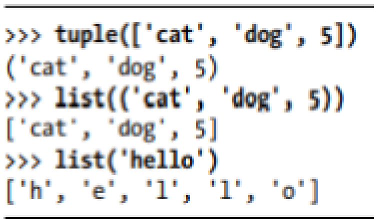
```
temp=[10,20,'cat',30,'apple',50]
print(temp[2:5])            ['cat',30,'apple']
print(temp[5:2])            error
print(temp[2:5:2])          ['cat','apple']
print(temp[:2])             [10,20]
print(temp[3:])             [30,'apple',50]
print(temp[0:-1])           [10,20,'cat',30,'apple',50]
print(temp[:])   `          [10,20,'cat',30,'apple',50]
print(temp[::-1])           [50,'apple',30,'cat',20,10]
```

6.a) The tuple data type is almost identical to the list data type, except in two ways.
First, tuples are typed with parentheses, (and), instead of square brackets, [and].
Second, benefit of using tuples instead of lists is that, because they are immutable and their contents don't change. Tuples cannot have their values modified, appended, or removed.



```
>>> eggs = ('hello', 42, 0.5)
>>> eggs[1] = 99
Traceback (most recent call last):
  File "<pyshell#5>", line 1, in <module>
    eggs[1] = 99
TypeError: 'tuple' object does not support item assignment
```

The functions list() and tuple() will return list and tuple versions of the values passed to them.



```
>>> tuple(['cat', 'dog', 5])
('cat', 'dog', 5)
>>> list(('cat', 'dog', 5))
['cat', 'dog', 5]
>>> list('hello')
['h', 'e', 'l', 'l', 'o']
```

6.b)

```
num=[45,22,14,65,97,72]
for  i in range(0,len(num)):
        if(num[i]%3==0 and num[i]%5==0):
                num[i]=”pppqqq”
        elif(num[i]%3==0):
                num[i]=”ppp”
        elif(num[i]%5==0):
                num[i]=”qqq”
        else:
           continue
print(num)
```