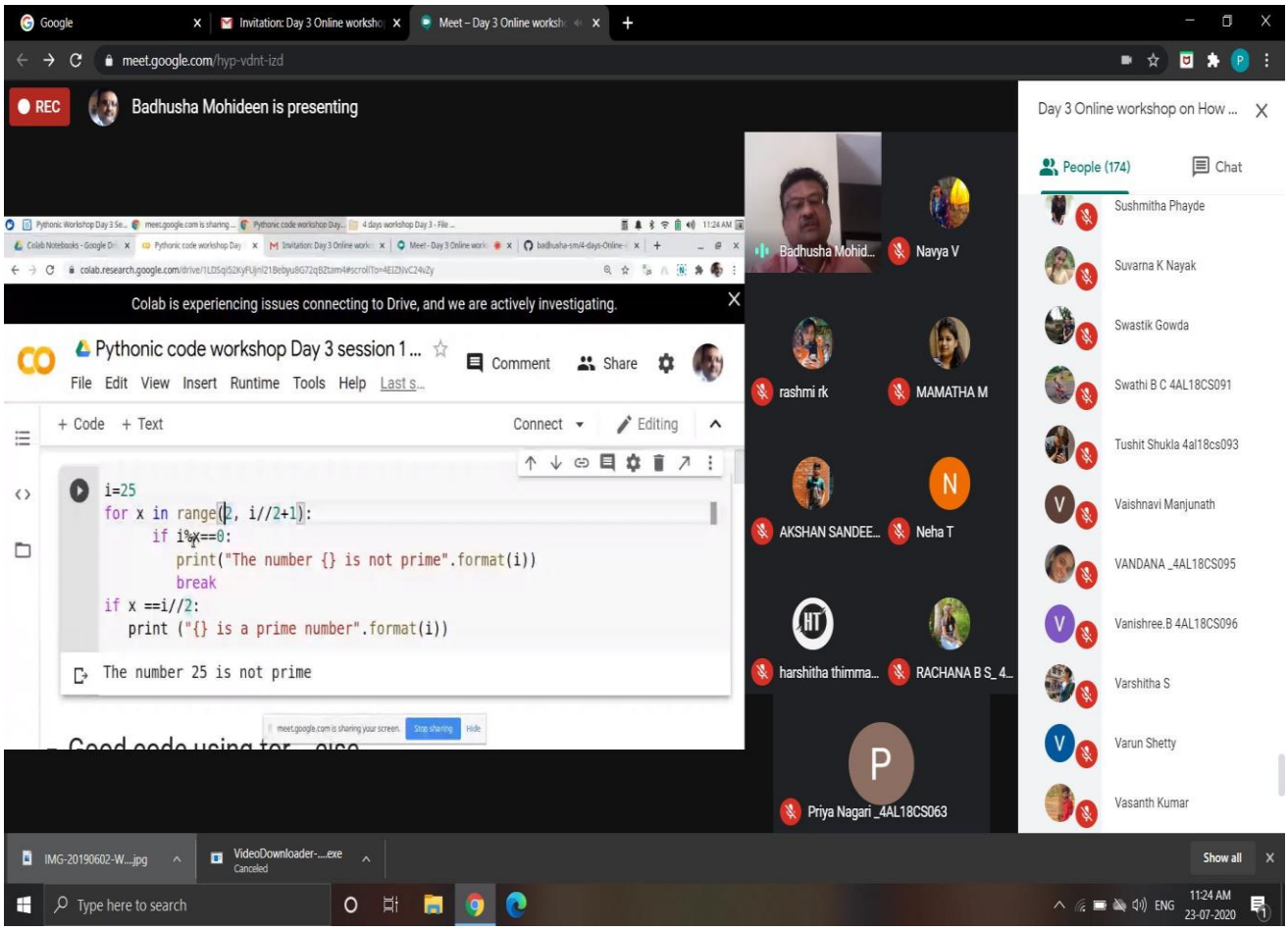
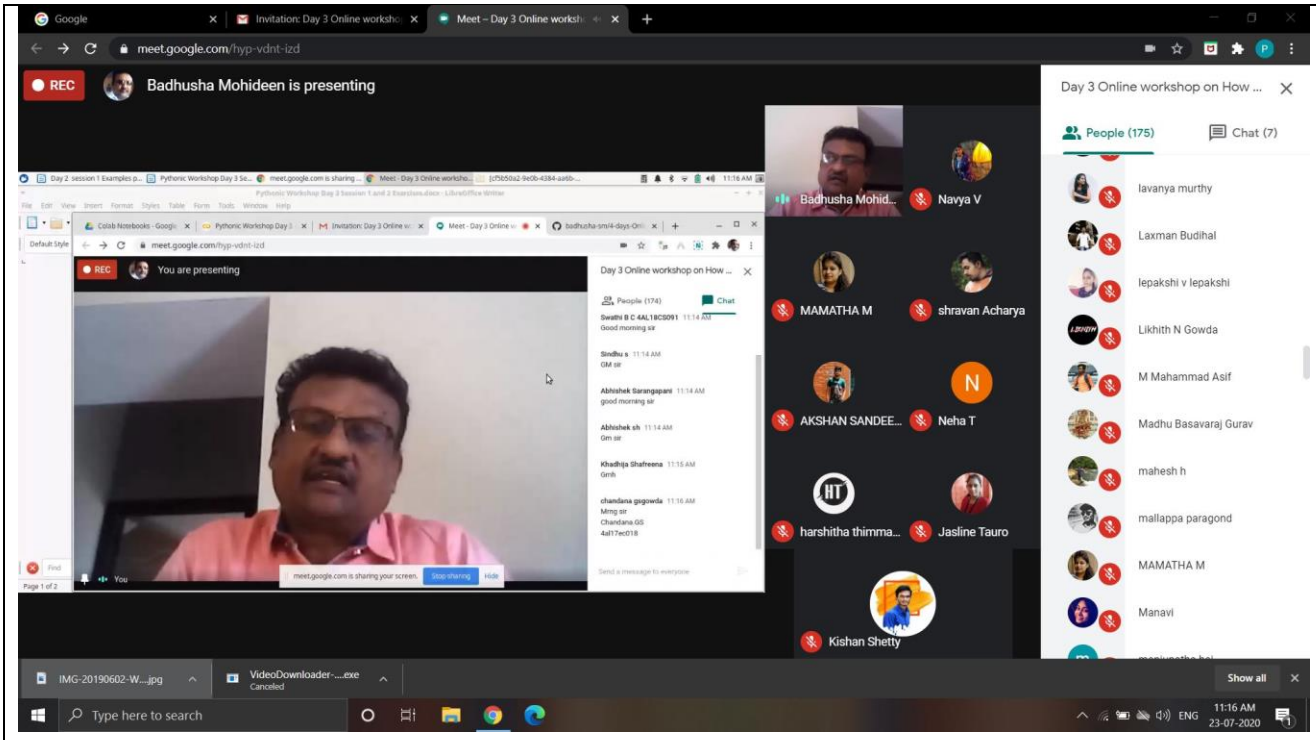
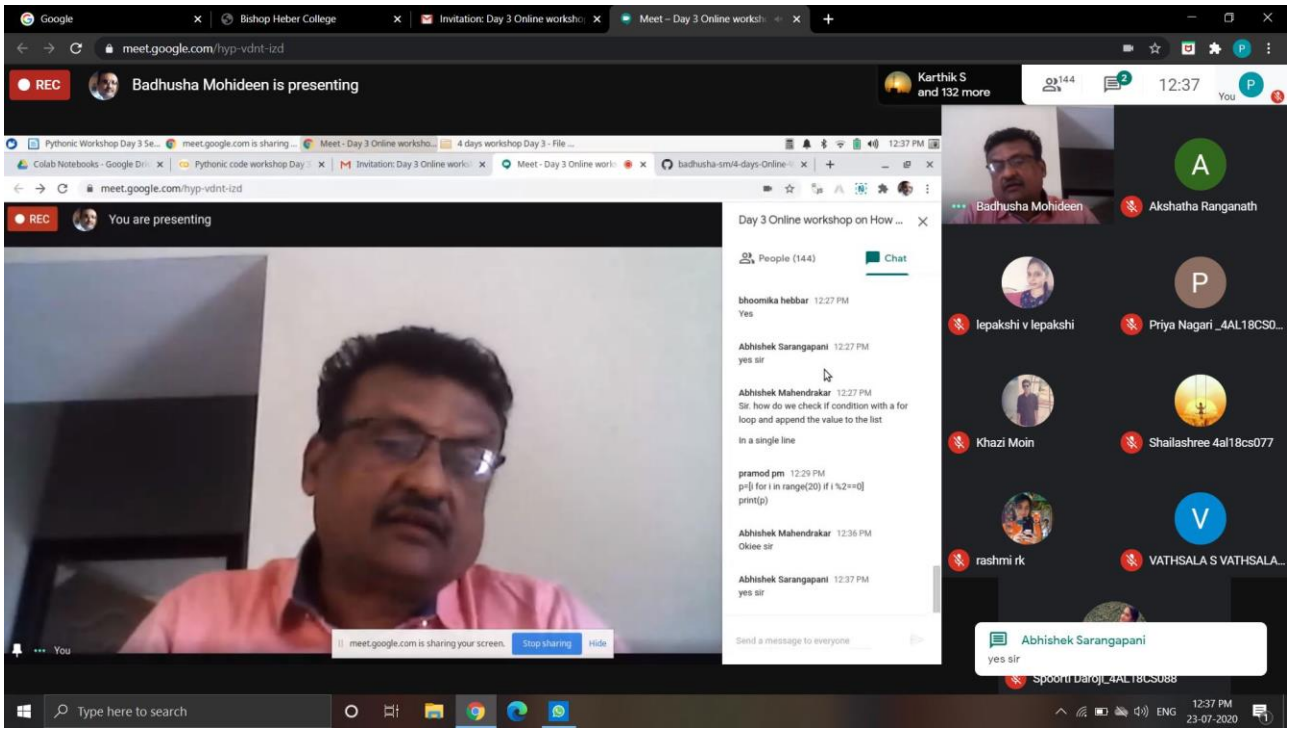
DAILY ASSESSMENT FORMAT

Date:	23 JULY 2020	Name:	PAVITHRAN S
Course:	PYTHON	USN:	4AL17EC068
Topic:	PYTHONIC CHALLENGE	Semester & Section:	6TH B
Github Repository:	Pavithran		

FORENOON SESSION DETAILS
Image of session




Google
Invitation: Day 3 Online worksho
Meet – Day 3 Online worksh
meet.google.com/hyp-vdnt-izd
REC
Badhusha Mohideen is presenting
Day 3 Online workshop on How ...
People (175)
Chat (7)
Badhusha Mohid...
Navya V
MAMATHA M
shravan Acharya
AKSHAN SANDEE...
Neha T
harshitha thimma...
Jasline Tauro
Kishan Shetty
lavanya murthy
Laxman Budihal
lepakshi v lepakshi
Likhith N Gowda
M Mahammad Asif
Madhu Basavaraj Gurav
mahesh h
mallappa paragond
MAMATHA M
Manavi
You are presenting
meet.google.com is sharing your screen.
Stop sharing
Hide
IMG-20190602-W....jpg
VideoDownloader-....exe
Canceled
Show all
Type here to search
ENG
11:16 AM
23-07-2020


Google
Bishop Heber College
Invitation: Day 3 Online worksho
Meet – Day 3 Online worksh
meet.google.com/hyp-vdnt-izd
REC
Badhusha Mohideen is presenting
Karthik S
and 132 more
12:37
You
You are presenting
Day 3 Online workshop on How ...
People (144)
Chat
bhoomika hebbar 12:27 PM
Yes
Abhishek Sarangapani 12:27 PM
yes sir
Abhishek Mahendrakar 12:27 PM
Sir, how do we check if condition with a for loop and append the value to the list
In a single line
pramod pm 12:29 PM
p=[i for i in range(20) if i %2==0]
print(p)
Abhishek Mahendrakar 12:36 PM
Okiee sir
Abhishek Sarangapani 12:37 PM
yes sir
Send a message to everyone
meet.google.com is sharing your screen.
Stop sharing
Hide
Badhusha Mohideen
Akshatha Ranganath
lepakshi v lepakshi
Priya Nagari _4AL18CS0...
Khazi Moin
Shailashree 4al18cs077
rashmi rk
VATHSALA S VATHSALA...
Abhishek Sarangapani
yes sir
Type here to search
ENG
12:37 PM
23-07-2020

Report – Report can be typed or hand written for up to two pages.

Check if a variable equals a constant

You don't need to explicitly compare a value to True, or None, or 0 – you can just add it to the if statement. See Truth Value Testing for a list of what is considered false.

Bad:

```
if attr == True:
    print 'True!'


if attr == None:
    print 'attr is None!'
```

Good:

```
# Just check the value
if attr:
    print 'attr is truthy!'


# or check for the opposite
if not attr:
    print 'attr is falsey!'


# or, since None is considered false, explicitly check for it
if attr is None:
    print 'attr is None!'
```

Access a Dictionary Element

Don't use the `dict.has_key()` method. Instead, use `x in d` syntax, or pass a default argument to `dict.get()`.

Bad:

```
d = {'hello': 'world'}

if d.has_key('hello'):

    print d['hello']    # prints 'world'

else:

    print 'default_value'
```

Good:

```
d = {'hello': 'world'}



print d.get('hello', 'default_value') # prints 'world'

print d.get('thingy', 'default_value') # prints 'default_value'



# Or:

if 'hello' in d:

    print d['hello']
```

Short Ways to Manipulate Lists

List comprehensions provide a powerful, concise way to work with lists.

Generator expressions follow almost the same syntax as list comprehensions but return a generator instead of a list.

Creating a new list requires more work and uses more memory. If you are just going to loop through the new list, prefer using an iterator instead.

Bad:

```
# needlessly allocates a list of all (gpa, name) entires in memory

valedictorian = max([(student.gpa, student.name) for student in graduates])
```

Good:

```
valedictorian = max((student.gpa, student.name) for student in graduates)
```

Use list comprehensions when you really need to create a second list, for example if you need to use the result multiple times.

If your logic is too complicated for a short list comprehension or generator expression, consider using a generator function instead of returning a list.

Good:

```
def make_batches(items, batch_size):
    """

    >>> list(make_batches([1, 2, 3, 4, 5], batch_size=3))

    [[1, 2, 3], [4, 5]]

    """

    current_batch = []

    for item in items:

        current_batch.append(item)

        if len(current_batch) == batch_size:

            yield current_batch

            current_batch = []

    yield current_batch
```

Never use a list comprehension just for its side effects.

Bad:

```
[print(x) for x in sequence]
```

Good:

```
for x in sequence:
    print(x)
```

Filtering a list

Bad:

Never remove items from a list while you are iterating through it.

```
# Filter elements greater than 4
a = [3, 4, 5]
for i in a:
    if i > 4:
        a.remove(i)
```

Don't make multiple passes through the list.

```
while i in a:
    a.remove(i)
```

Good:

Use a list comprehension or generator expression.

```
# comprehensions create a new list object
filtered_values = [value for value in sequence if value != x]


# generators don't create another list
filtered_values = (value for value in sequence if value != x)
```

Possible side effects of modifying the original list

Modifying the original list can be risky if there are other variables referencing it. But you can use *slice assignment* if you really want to do that.

```
# replace the contents of the original list

sequence[::] = [value for value in sequence if value != x]
```

Modifying the values in a list

Bad:

Remember that assignment never creates a new object. If two or more variables refer to the same list, changing one of them changes them all.

```
# Add three to all list members.

a = [3, 4, 5]

b = a                       # a and b refer to the same list object


for i in range(len(a)):

    a[i] += 3               # b[i] also changes
```

Good:

It's safer to create a new list object and leave the original alone.

```
a = [3, 4, 5]

b = a


# assign the variable "a" to a new list without changing "b"

a = [i + 3 for i in a]
```

Use `enumerate()` keep a count of your place in the list.

```
a = [3, 4, 5]

for i, item in enumerate(a):
```

```
    print i, item

# prints

# 0 3

# 1 4

# 2 5
```

The `enumerate()` function has better readability than handling a counter manually. Moreover, it is better optimized for iterators.

Read From a File

Use the `with open` syntax to read from files. This will automatically close files for you.

Bad:

```
f = open('file.txt')

a = f.read()

print a

f.close()
```

Good:

```
with open('file.txt') as f:

    for line in f:

        print line
```

The `with` statement is better because it will ensure you always close the file, even if an exception is raised inside the `with` block.

Line Continuations

When a logical line of code is longer than the accepted limit, you need to split it over multiple physical lines. The Python interpreter will join consecutive lines if the last character of the line is a backslash. This is helpful in some cases, but should usually be avoided because of its fragility: a white space added to the end of the line, after the

backslash, will break the code and may have unexpected results.

A better solution is to use parentheses around your elements. Left with an unclosed parenthesis on an end-of-line the Python interpreter will join the next line until the parentheses are closed. The same behavior holds for curly and square braces.

Bad:

```
my_very_big_string = """For a long time I used to go to bed early. Sometimes, \

    when I had put out my candle, my eyes would close so quickly that I had not even \

    time to say “I’m going to sleep.”"""


from some.deep.module.inside.a.module import a_nice_function, another_nice_function, \

    yet_another_nice_function
```

Good:

```
my_very_big_string = (

    "For a long time I used to go to bed early. Sometimes, "

    "when I had put out my candle, my eyes would close so quickly "

    "that I had not even time to say “I’m going to sleep.”"

)


from some.deep.module.inside.a.module import (

    a_nice_function, another_nice_function, yet_another_nice_function)
```