Code Segments that Perform Iteration, Troubleshooting and Error Handling & Function

Pandas Group – Data Science Track B

My Edu Solve

Ajie Prasetya | Dewa Ayu Rai I.M. | M. Ramadhoni | Firdausi Nurus Sa'idah | Vivi Indah Fitriani | I K. Juni Saputra

Python Material Discussion

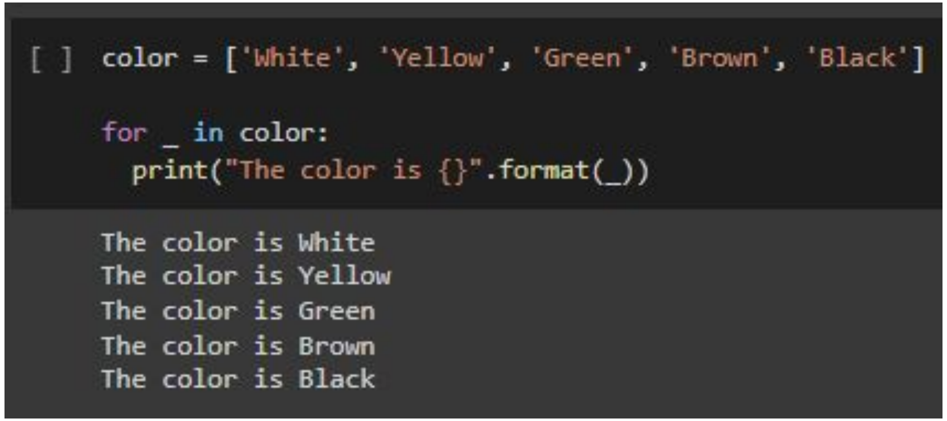
For and else, while and else, Break, Continue, Pass & List Comprehension

Error and Exception Handling

Function, Return, Pass by reference vs value and Argument & Parameter

For

For is a function that can loop over each type of variable in the form of a collection or sequence (list, string, or range). If a list or sequence contains an expression, it will evaluate it first. Then the first item in the sequence/list will be assigned as an iterating_var variable, and then it will continue to the next item until it runs out.

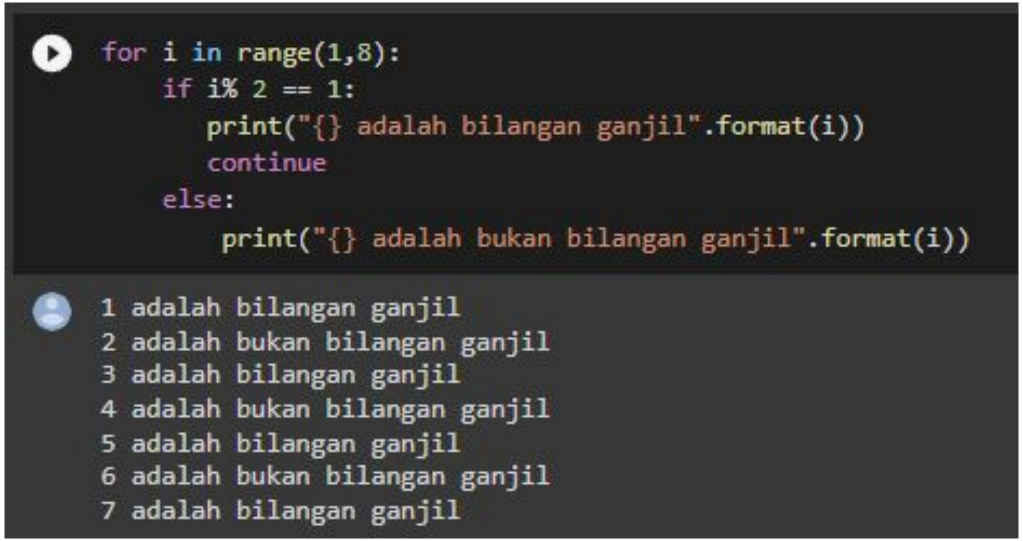


```
[ ] color = ['White', 'Yellow', 'Green', 'Brown', 'Black']

    for _ in color:
      print("The color is {}".format(_))
```

```
The color is White
The color is Yellow
The color is Green
The color is Brown
The color is Black
```

Else After For

The else function after for is a function that takes precedence over search loops to provide a way out of the program when the search is not found

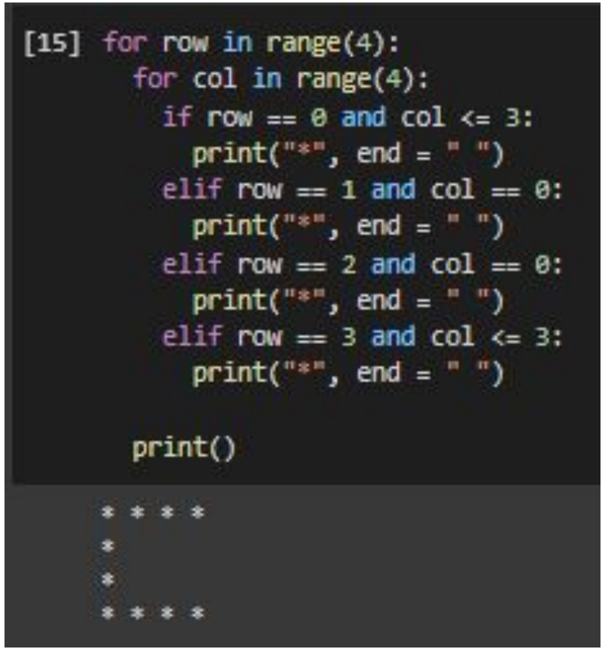


```
for i in range(1,8):
    if i% 2 == 1:
       print("{} adalah bilangan ganjil".format(i))
       continue
    else:
        print("{} adalah bukan bilangan ganjil".format(i))
```

```
1 adalah bilangan ganjil
2 adalah bukan bilangan ganjil
3 adalah bilangan ganjil
4 adalah bukan bilangan ganjil
5 adalah bilangan ganjil
6 adalah bukan bilangan ganjil
7 adalah bilangan ganjil
```

Nested Loop

A nested loop is a loop inside the body of the outer loop. The inner or outer loop can be any type, such as a while loop or for loop. For example, the outer for loop can contain a while loop and vice versa.

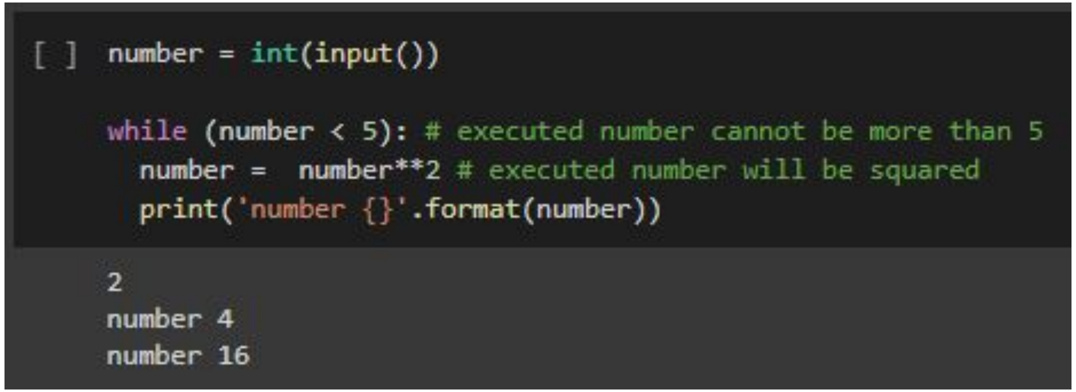


```
[15] for row in range(4):
       for col in range(4):
         if row == 0 and col <= 3:
           print("*", end = " ")
         elif row == 1 and col == 0:
           print("*", end = " ")
         elif row == 2 and col == 0:
           print("*", end = " ")
         elif row == 3 and col <= 3:
           print("*", end = " ")

       print()
```

```
* * * *
*
*
* * * *
```

While

While in Python is used to execute statements as long as the condition as long as the test expression (condition) is true. The condition can be any expression and includes all non-zero values. When the condition becomes False, the program will continue to the line after the statement block.

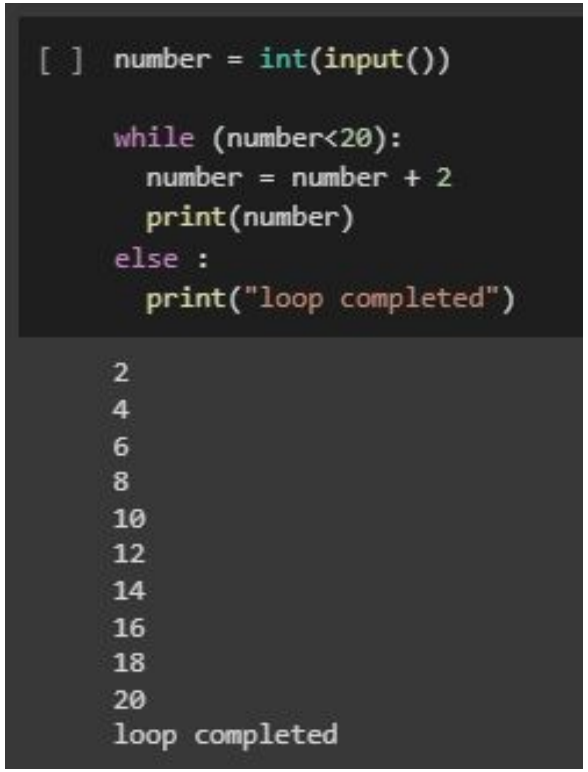


```
[ ]  number = int(input())

     while (number < 5): # executed number cannot be more than 5
       number =  number**2 # executed number will be squared
       print('number {}'.format(number))
```

```
2
number 4
number 16
```

Else After While

In the while statement, the else blok statement will always be executed when the while condition is false

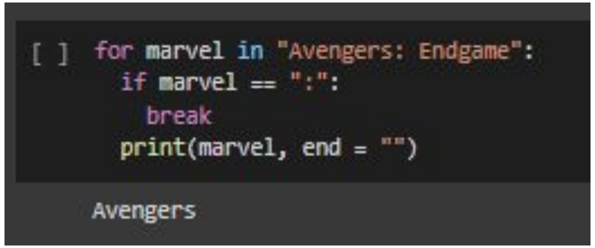


```
[ ]  number = int(input())

     while (number<20):
       number = number + 2
       print(number)
     else :
       print("loop completed")
```

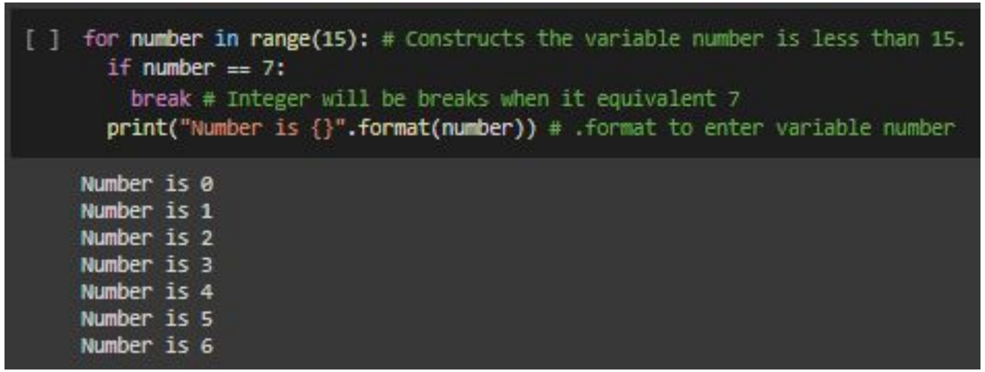
```
2
4
6
8
10
12
14
16
18
20
loop completed
```

Break

The break statement ends the loop containing it. Program control flows to the statement immediately after the loop body. If the break statement is inside a nested loop (a loop inside another loop), the break statement ends the loop according to the level or in which loop it is.



```
[ ]  for marvel in "Avengers: Endgame":
       if marvel == ":":
         break
       print(marvel, end = "")

     Avengers
```



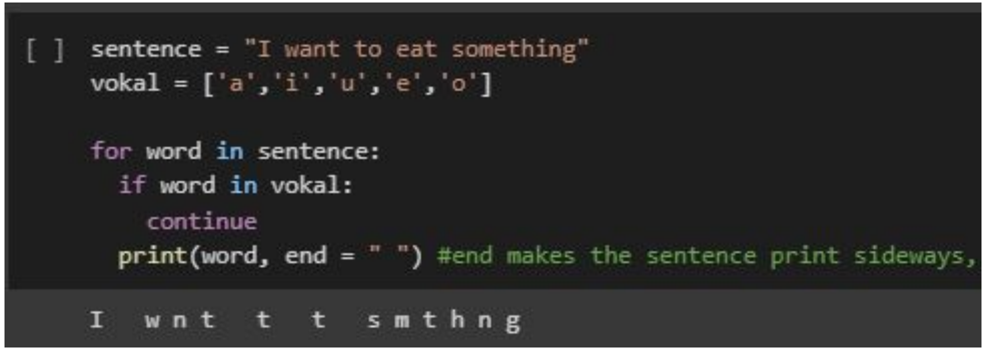
```
[ ]  for number in range(15): # Constructs the variable number is less than 15.
       if number == 7:
         break # Integer will be breaks when it equivalent 7
       print("Number is {}".format(number)) # .format to enter variable number

     Number is 0
     Number is 1
     Number is 2
     Number is 3
     Number is 4
     Number is 5
     Number is 6
```

Example
1 and 2

Continue

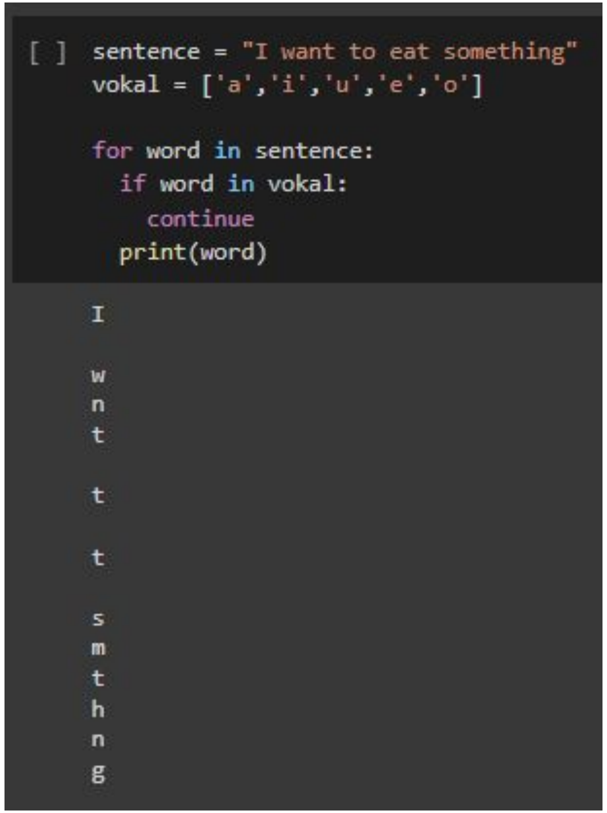
The continue statement is used to skip the rest of the code (statement) inside a loop for the current iteration only. Loop does not terminate but continues on with the next iteration.

print sideways



```
[ ] sentence = "I want to eat something"
    vokal = ['a','i','u','e','o']

    for word in sentence:
      if word in vokal:
        continue
      print(word, end = " ") #end makes the sentence print sideways,
```

```
    I  w n t  t  t  s m t h n g
```

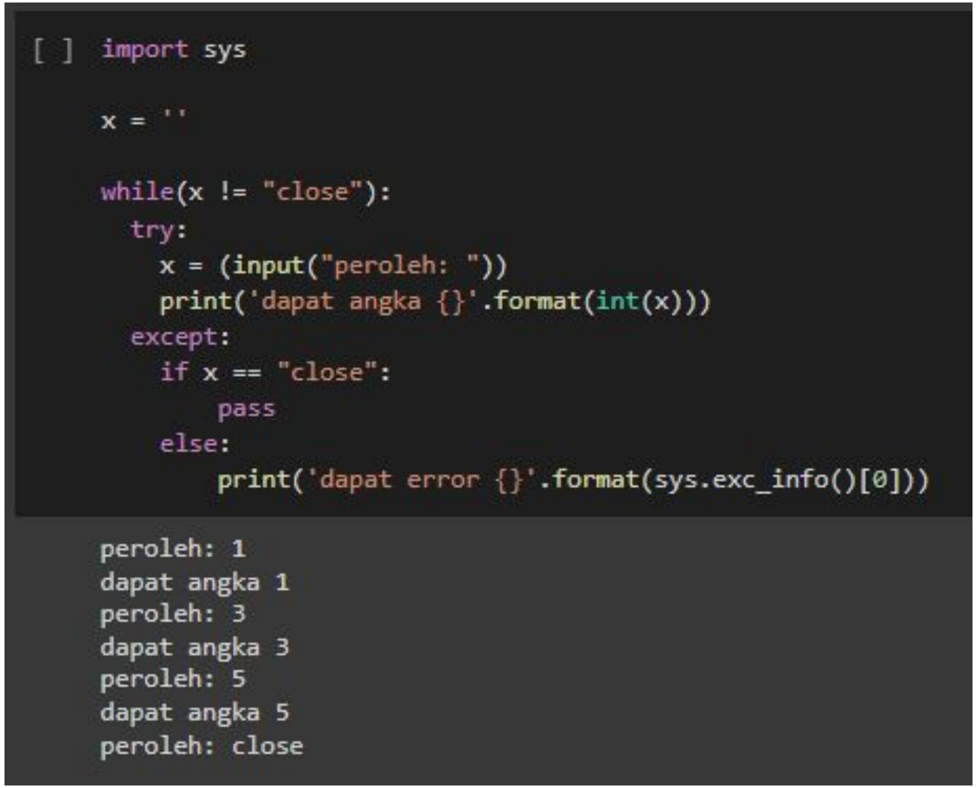


```
[ ] sentence = "I want to eat something"
    vokal = ['a','i','u','e','o']

    for word in sentence:
      if word in vokal:
        continue
      print(word)
```

```
    I

    w
    n
    t

    t

    t

    s
    m
    t
    h
    n
    g
```

Pass

The pass statement is used as a placeholder for future code. When the pass statement is executed, nothing happens, but you avoid getting an error when empty code is not allowed. The pass statement is a null(empty) operation, nothing happens when the pass is executed.

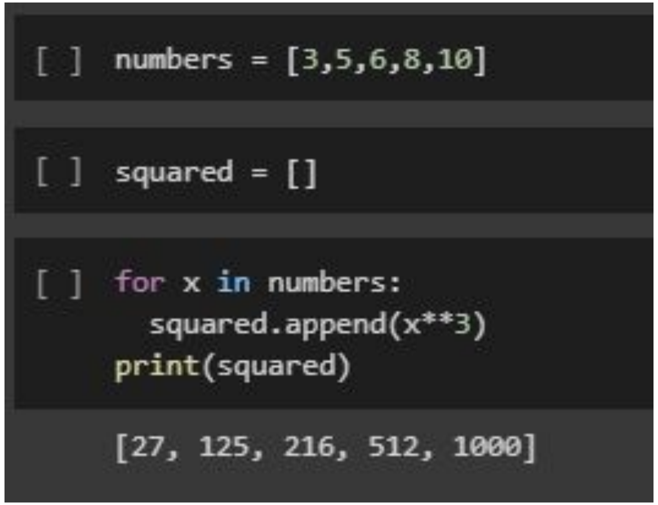


```
[ ]  import sys

     x = ''

     while(x != "close"):
       try:
         x = (input("peroleh: "))
         print('dapat angka {}'.format(int(x)))
       except:
         if x == "close":
             pass
         else:
             print('dapat error {}'.format(sys.exc_info()[0]))
```

```
     peroleh: 1
     dapat angka 1
     peroleh: 3
     dapat angka 3
     peroleh: 5
     dapat angka 5
     peroleh: close
```

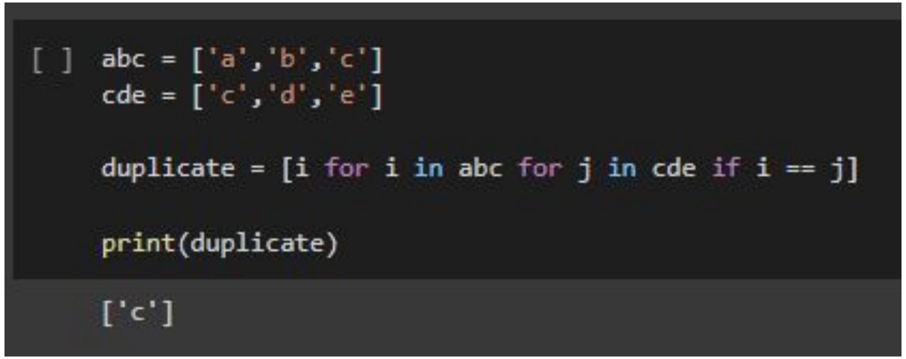
List Comprehension

List comprehensions are used for creating new lists from other iterables like tuples, strings, arrays, lists, etc. A list comprehension consists of brackets containing the expression, which is executed for each element along with the for loop to iterate over each element



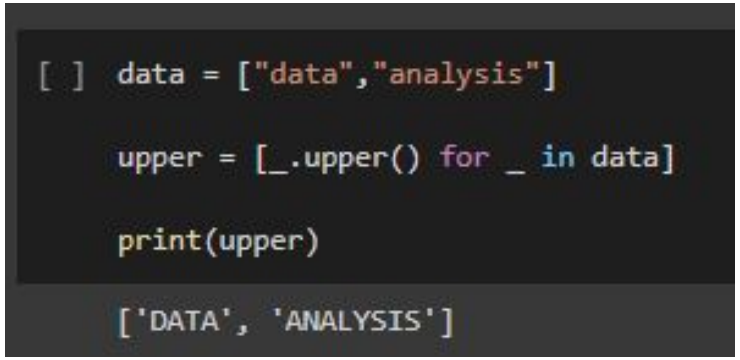
```
[ ]  numbers = [3,5,6,8,10]

[ ]  squared = []

[ ]  for x in numbers:
       squared.append(x**3)
     print(squared)

     [27, 125, 216, 512, 1000]
```

Different Writing List Comprehension



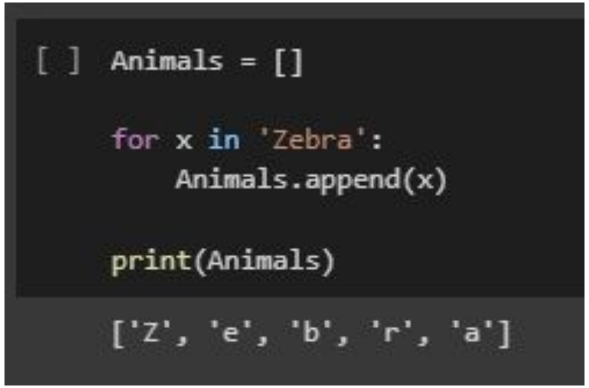
```
[ ]  abc = ['a','b','c']
     cde = ['c','d','e']

     duplicate = [i for i in abc for j in cde if i == j]

     print(duplicate)

     ['c']
```

Add upper() function to list comprehension



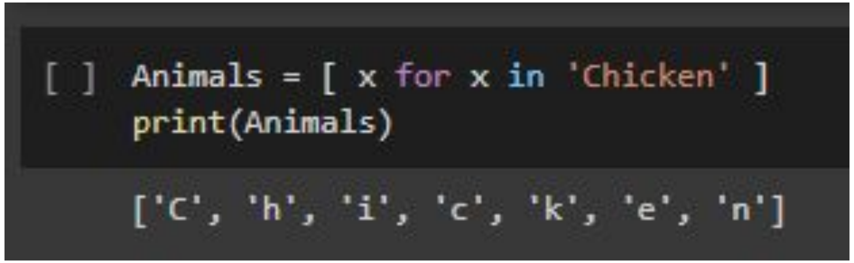
```
[ ]  data = ["data","analysis"]

     upper = [_.upper() for _ in data]

     print(upper)

     ['DATA', 'ANALYSIS']
```

List Comprehension (List With Inline Loop and If)



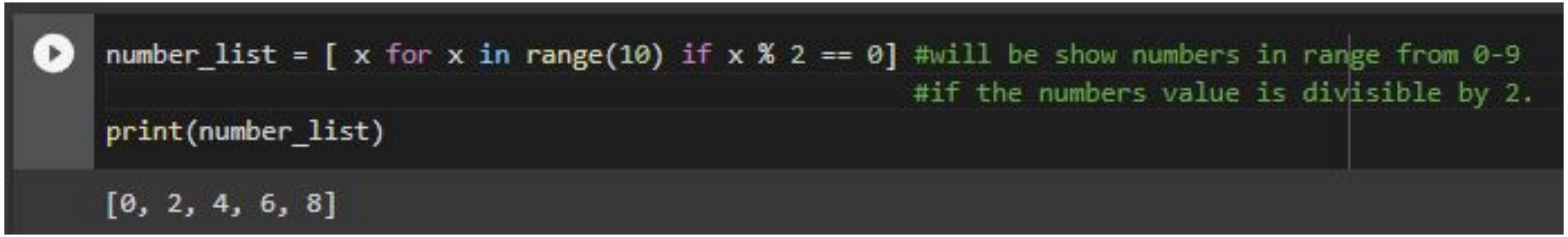
```
[ ] Animals = []

    for x in 'Zebra':
        Animals.append(x)

    print(Animals)

    ['Z', 'e', 'b', 'r', 'a']
```

You can also modify the syntax so it can be like this



```
[ ] Animals = [ x for x in 'Chicken' ]
    print(Animals)

    ['C', 'h', 'i', 'c', 'k', 'e', 'n']
```

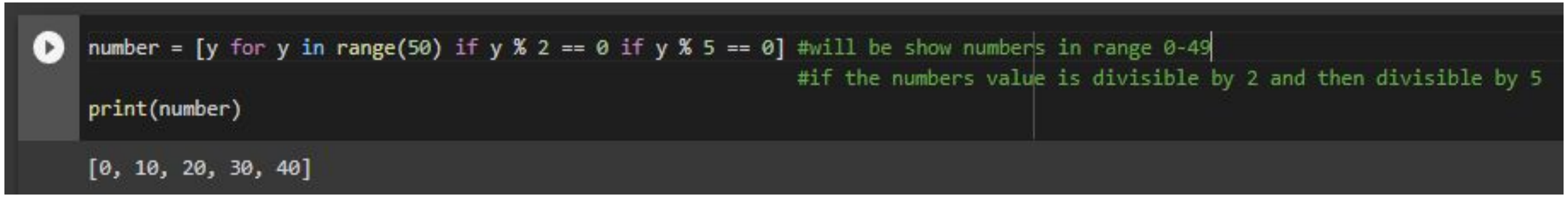
Inline Loop and If



```
number_list = [ x for x in range(10) if x % 2 == 0] #will be show numbers in range from 0-9
                                                     #if the numbers value is divisible by 2.
print(number_list)
```

```
[0, 2, 4, 6, 8]
```

And



```
number = [y for y in range(50) if y % 2 == 0 if y % 5 == 0] #will be show numbers in range 0-49
                                                            #if the numbers value is divisible by 2 and then divisible by 5
print(number)
```

```
[0, 10, 20, 30, 40]
```

Error and Exception Handling

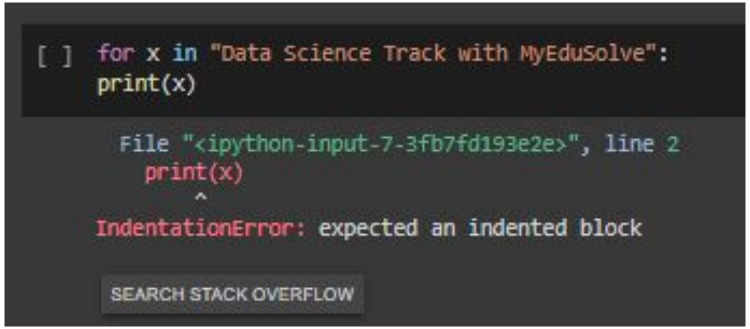
There are two types of errors, that is

1. Syntax Errors,
2. Exceptions

Syntax Error

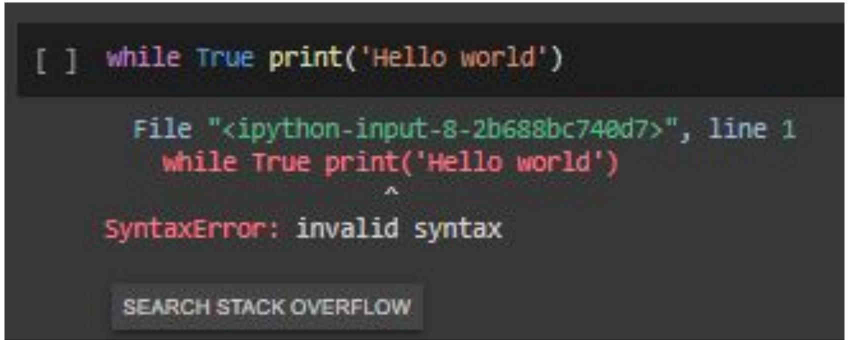
It usually happens when python does not understand what you are commanding and generally in people are still just learning python.

the following example will show one of the errors that is the **incorrect placement** of the **indent**

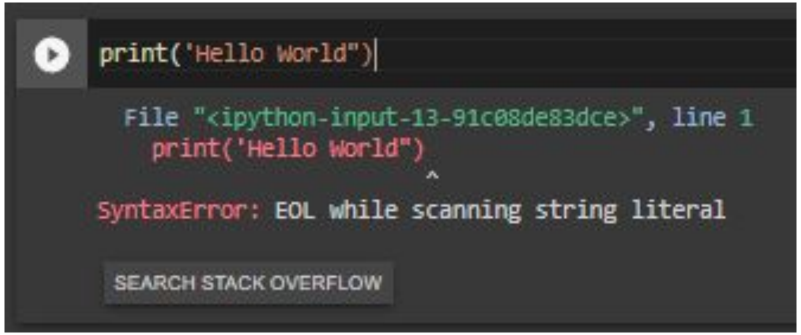


```
[ ]  for x in "Data Science Track with MyEduSolve":
     print(x)

       File "<ipython-input-7-3fb7fd193e2e>", line 2
         print(x)
             ^
     IndentationError: expected an indented block
```

The following example shows syntax writing errors



```
[ ] while True print('Hello world')

  File "<ipython-input-8-2b688bc740d7>", line 1
    while True print('Hello world')
                   ^
SyntaxError: invalid syntax
```

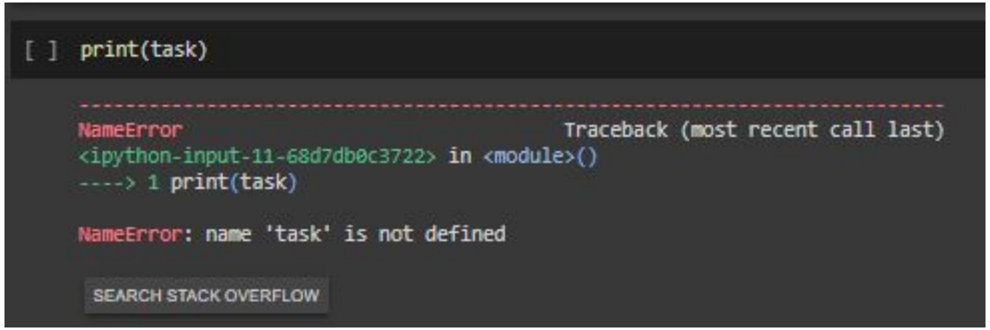


```
print('Hello World")

  File "<ipython-input-13-91c08de83dce>", line 1
    print('Hello World")
                        ^
SyntaxError: EOL while scanning string literal
```

Exceptions

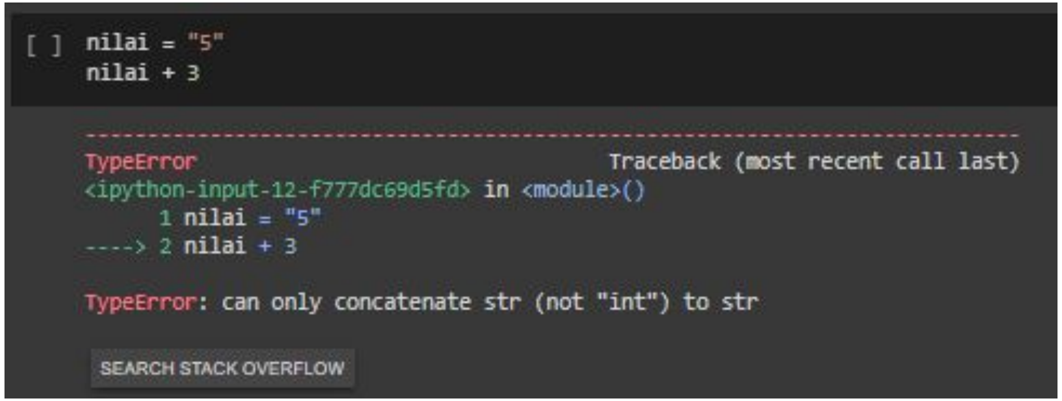
Exceptions can occur when an error occurs when the command we entered is executed and will be fatal if not addressed immediately.

A simple example of this error when we forget to specify a variable but we call the variable



```
[ ]  print(task)

     ---------------------------------------------------------------------------
     NameError                                 Traceback (most recent call last)
     <ipython-input-11-68d7db0c3722> in <module>()
     ----> 1 print(task)

     NameError: name 'task' is not defined
```

Next example when we operate between integer type variables with strings

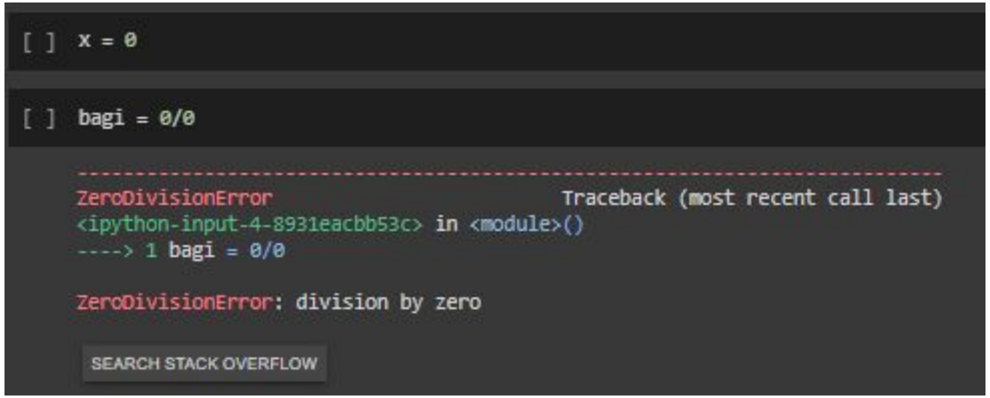


```
[ ] nilai = "5"
    nilai + 3
```

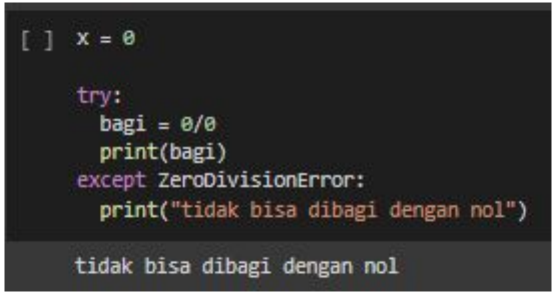
```
---------------------------------------------------------------------------
TypeError                                 Traceback (most recent call last)
<ipython-input-12-f777dc69d5fd> in <module>()
      1 nilai = "5"
----> 2 nilai + 3

TypeError: can only concatenate str (not "int") to str
```

Exception Handling

The exception handling process uses a try statement paired with except. For example we want to handle an **exception** that occurs if there is a **division of a number** by the **value 0 (0)**

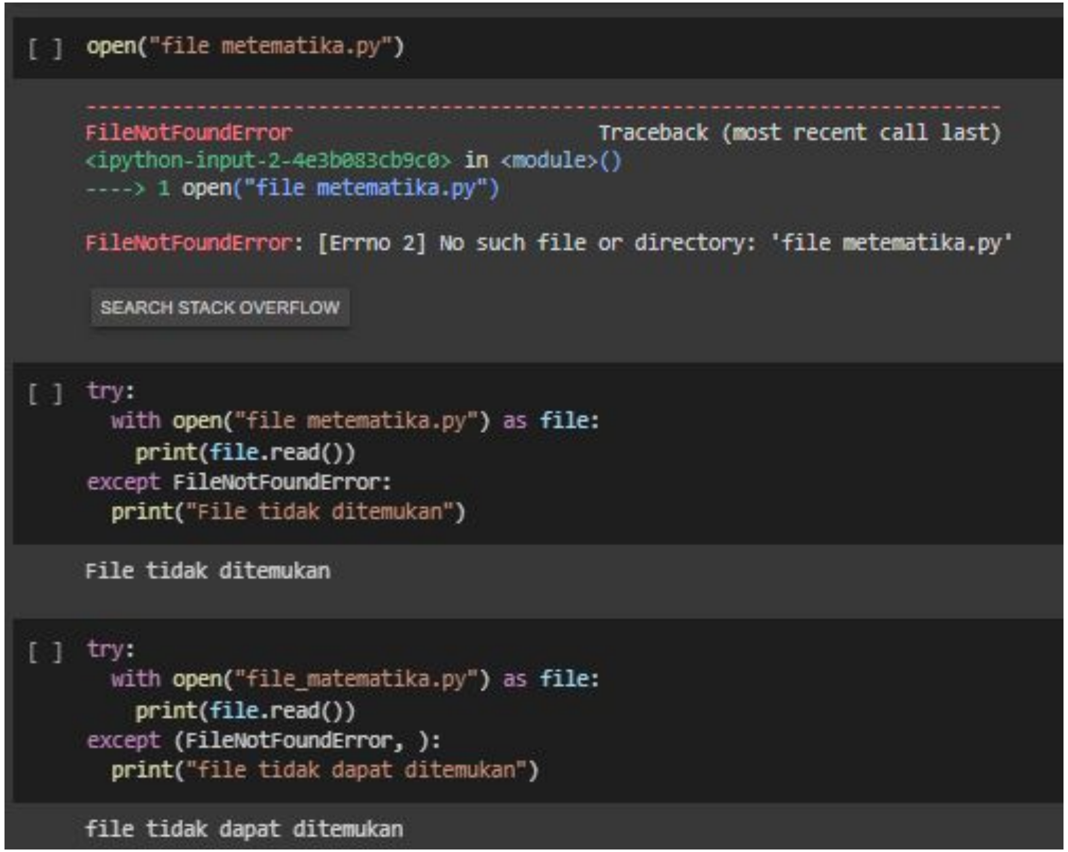


```
[ ] x = 0

[ ] bagi = 0/0

    ---------------------------------------------------------------------------
    ZeroDivisionError                         Traceback (most recent call last)
    <ipython-input-4-8931eacbb53c> in <module>()
    ----> 1 bagi = 0/0

    ZeroDivisionError: division by zero
```



```
[ ] x = 0

    try:
      bagi = 0/0
      print(bagi)
    except ZeroDivisionError:
      print("tidak bisa dibagi dengan nol")

    tidak bisa dibagi dengan nol
```

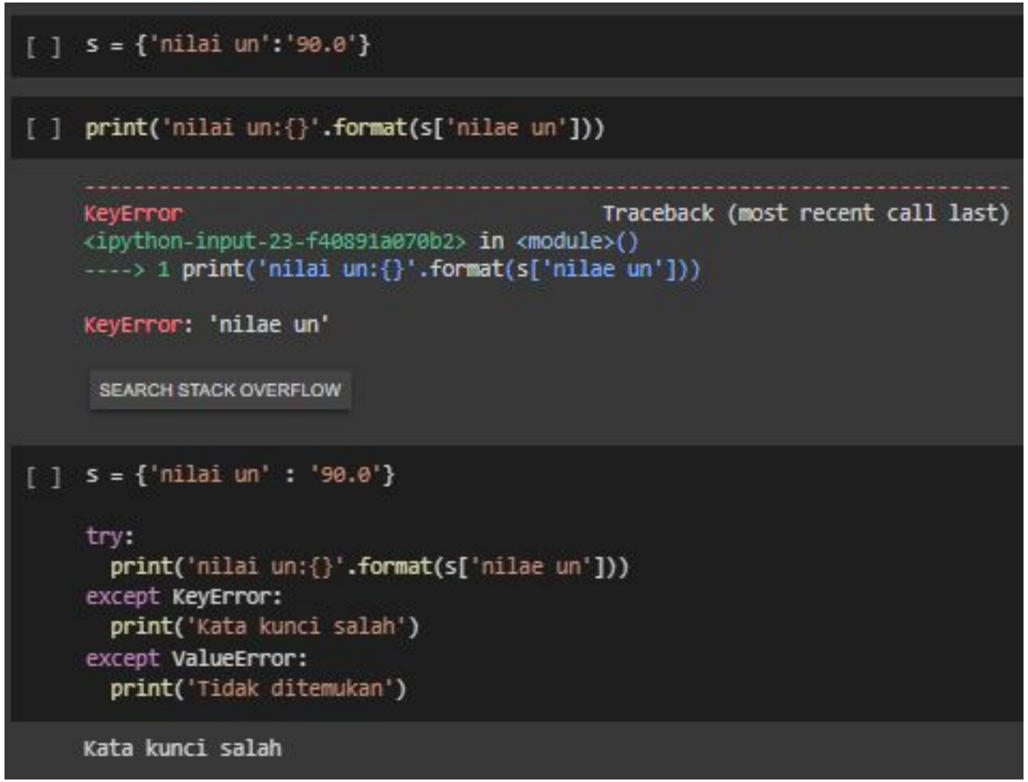
ZeroDivisionError is an exception that occurs when program execution results in a mathematical calculation of division by zero



```
[ ] open("file metematika.py")

    ---------------------------------------------------------------------------
    FileNotFoundError                         Traceback (most recent call last)
    <ipython-input-2-4e3b083cb9c0> in <module>()
    ----> 1 open("file metematika.py")

    FileNotFoundError: [Errno 2] No such file or directory: 'file metematika.py'
```

```
[ ] try:
      with open("file metematika.py") as file:
        print(file.read())
    except FileNotFoundError:
      print("File tidak ditemukan")

    File tidak ditemukan

[ ] try:
      with open("file_matematika.py") as file:
        print(file.read())
    except (FileNotFoundError, ):
      print("file tidak dapat ditemukan")

    file tidak dapat ditemukan
```

With a pair of try and except statements, the application does not stop but prints on the screen that the file was not found



```
[ ]  s = {'nilai un':'90.0'}
```

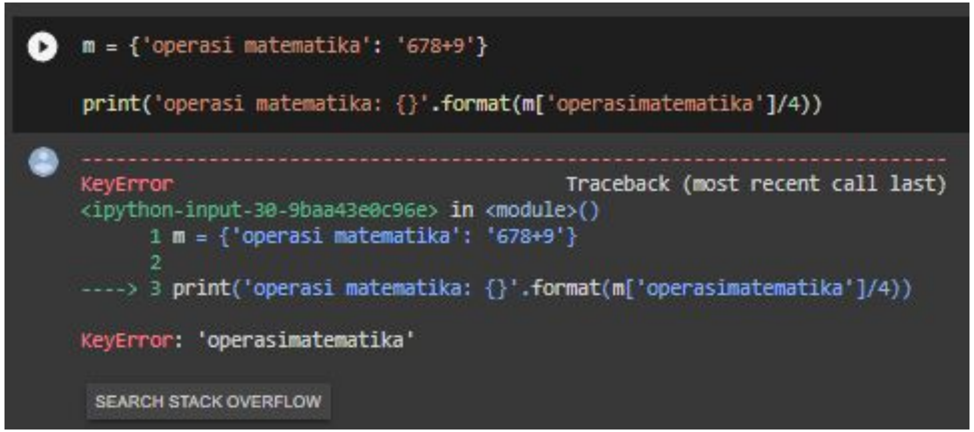
```
[ ]  print('nilai un:{}'.format(s['nilae un']))
```

```
---------------------------------------------------------------------------
KeyError                                  Traceback (most recent call last)
<ipython-input-23-f40891a070b2> in <module>()
----> 1 print('nilai un:{}'.format(s['nilae un']))

KeyError: 'nilae un'
```

```
[ ]  s = {'nilai un' : '90.0'}

     try:
       print('nilai un:{}'.format(s['nilae un']))
     except KeyError:
       print('Kata kunci salah')
     except ValueError:
       print('Tidak ditemukan')
```

```
Kata kunci salah
```

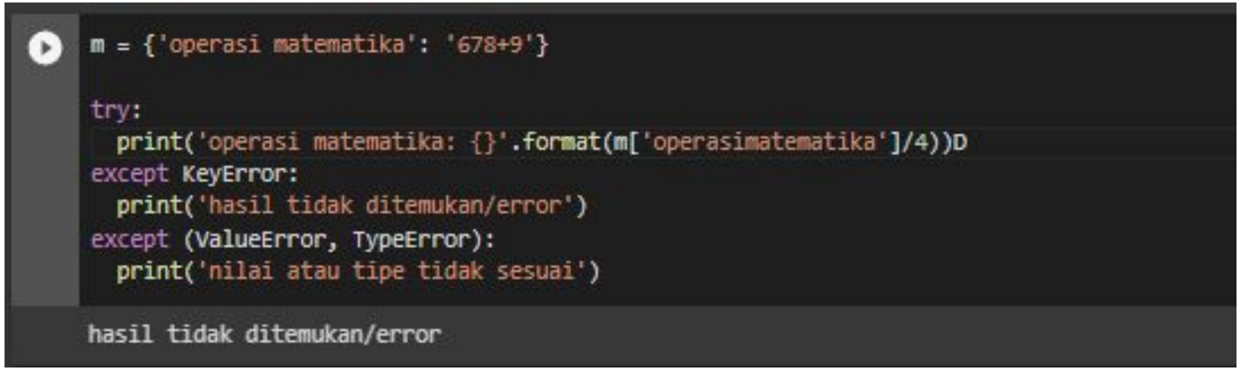
In more complex applications, exception handling can use a single except statement that handles more than one type of error concatenated in a tuple.



```
m = {'operasi matematika': '678+9'}

print('operasi matematika: {}'.format(m['operasimatematika']/4))
```

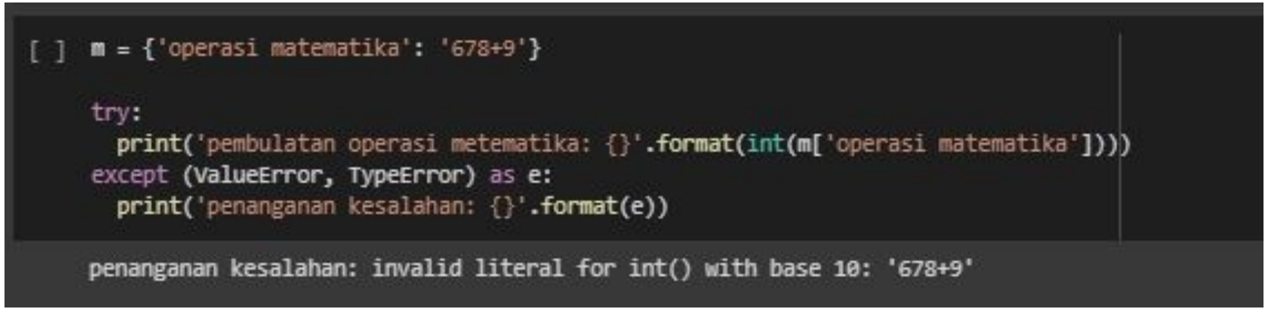
```
---------------------------------------------------------------------------
KeyError                                  Traceback (most recent call last)
<ipython-input-30-9baa43e0c96e> in <module>()
      1 m = {'operasi matematika': '678+9'}
      2 
----> 3 print('operasi matematika: {}'.format(m['operasimatematika']/4))

KeyError: 'operasimatematika'
```

Next>>



```
m = {'operasi matematika': '678+9'}

try:
  print('operasi matematika: {}'.format(m['operasimatematika']/4))D
except KeyError:
  print('hasil tidak ditemukan/error')
except (ValueError, TypeError):
  print('nilai atau tipe tidak sesuai')
```

```
hasil tidak ditemukan/error
```

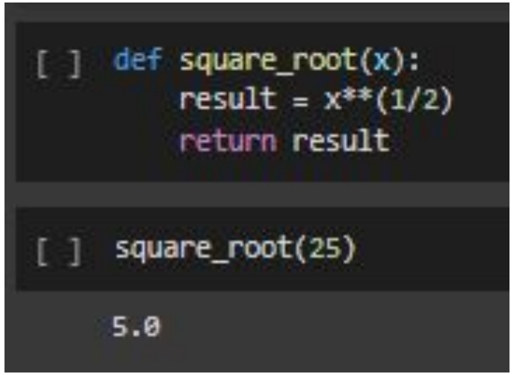


```
m = {'operasi matematika': '678+9'}

try:
  print('pembulatan operasi metematika: {}'.format(int(m['operasi matematika'])))
except (ValueError, TypeError) as e:
  print('penanganan kesalahan: {}'.format(e))
```

```
penanganan kesalahan: invalid literal for int() with base 10: '678+9'
```

Function

A function is a block of code that only runs when it is called. Python functions return a value using a return statement, if one is specified. A function can be called anywhere after the function has been declared.



```
[ ] def square_root(x):
        result = x**(1/2)
        return result
```

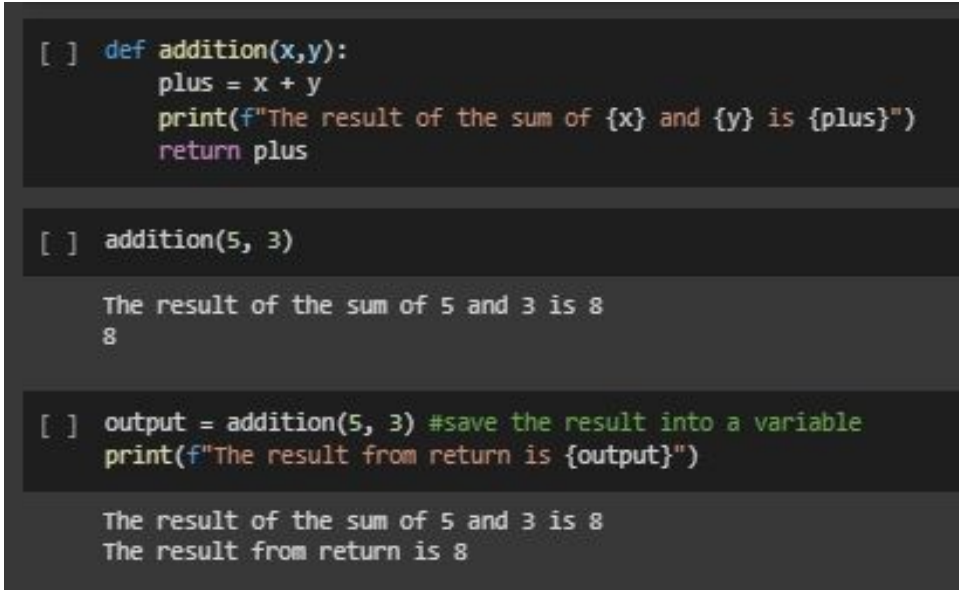
```
[ ] square_root(25)
```

```
5.0
```

Return

Return expression will make program execution exit the function

Example 1



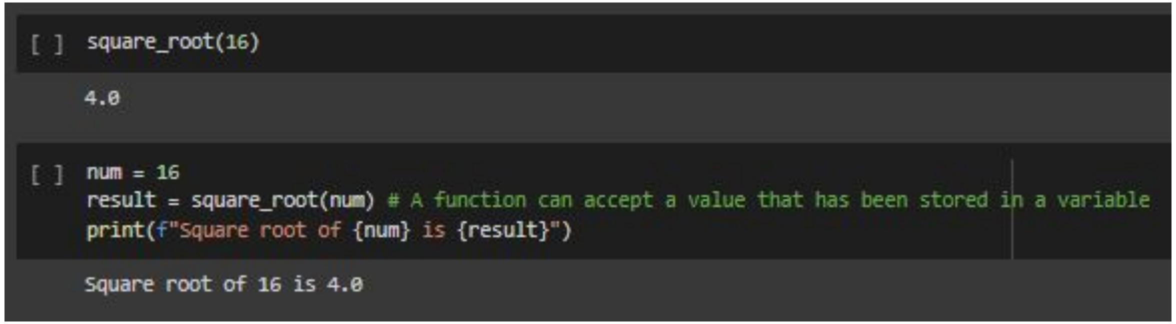
```
[ ] def addition(x,y):
        plus = x + y
        print(f"The result of the sum of {x} and {y} is {plus}")
        return plus
```

```
[ ] addition(5, 3)
```

```
The result of the sum of 5 and 3 is 8
8
```

```
[ ] output = addition(5, 3) #save the result into a variable
    print(f"The result from return is {output}")
```

```
The result of the sum of 5 and 3 is 8
The result from return is 8
```



```
[ ] square_root(16)
```

```
4.0
```

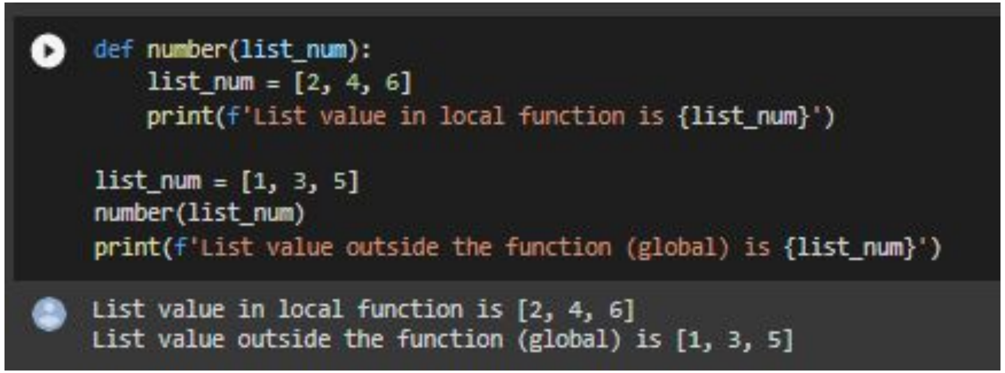
```
[ ] num = 16
    result = square_root(num) # A function can accept a value that has been stored in a variable
    print(f"Square root of {num} is {result}")
```

```
Square root of 16 is 4.0
```

Example 2

Pass by reference vs value

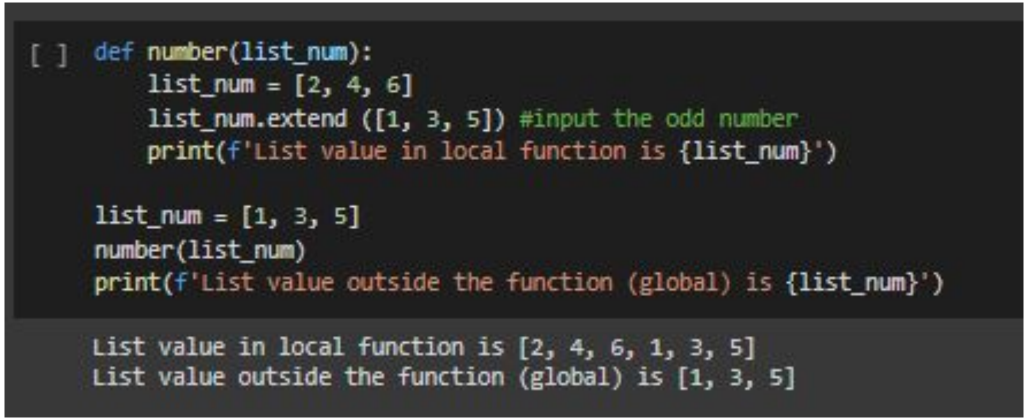
All parameters (arguments) in the python language are "passed by reference". This means that when we change a variable, the data that refers to it will also change, both inside the function and outside the calling function. Unless we perform an assignment operation that changes the reference parameter.



```python
def number(list_num):
    list_num = [2, 4, 6]
    print(f'List value in local function is {list_num}')

list_num = [1, 3, 5]
number(list_num)
print(f'List value outside the function (global) is {list_num}')
```

```
List value in local function is [2, 4, 6]
List value outside the function (global) is [1, 3, 5]
```

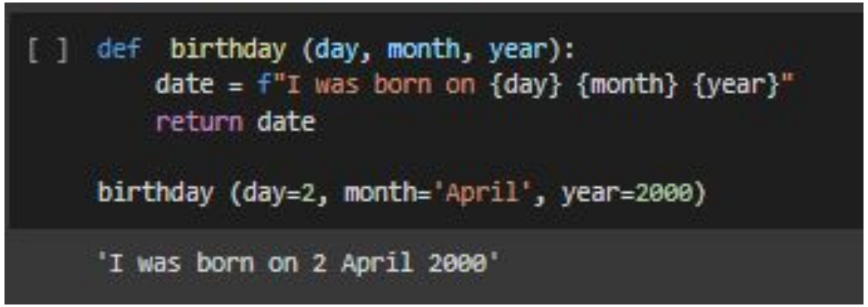
Next Example Pass by reference vs value>>



```
[ ] def number(list_num):
        list_num = [2, 4, 6]
        list_num.extend ([1, 3, 5]) #input the odd number
        print(f'List value in local function is {list_num}')

    list_num = [1, 3, 5]
    number(list_num)
    print(f'List value outside the function (global) is {list_num}')
```

```
    List value in local function is [2, 4, 6, 1, 3, 5]
    List value outside the function (global) is [1, 3, 5]
```

Argument and Parameter



```
[ ] def  birthday (day, month, year):
        date = f"I was born on {day} {month} {year}"
        return date

    birthday (day=2, month='April', year=2000)
```

```
    'I was born on 2 April 2000'
```

Thank You