MyEduSolve

Get to Know More About Python

Created by Sintaks Group

Table of Contents

Use Case Summary

Objective Statement :
Get insight into looping, break and continue statement, list comprehension, pass an error, syntax error and exceptions, error exceptions types, and function in python.

Challenges :
Syntax in python is case sensitive, Requires logic to solve problems in code, Requires logic to make list comprehension, Require sensitivity to syntax errors and exceptions, Requires attention to indentation, Requires a good understanding of local and global variables.

Methodology :
Syntax for looping, break and continue statement, list comprehensions, pass an error, error and exceptions, Try, Except, Concept function in python.

Use Case Summary

Benefit :
Understand to make a looping, break and continue statement, list comprehensions, pass an error, concept syntax error and exceptions, exceptions handling, and call a function, return statement, difference between argument and parameter, and pass by reference and pass by value in python.

Expected Outcome :
Knowing the syntax used to make looping, break and continue statement, list comprehension, concept syntax error and exceptions, perform exception handling, create and call a function, return a specified value, and pass information to a function.

Looping is the process of repeating the execution of one or more statement blocks without stopping, as long as the reference condition is met.

Usually a variable for iteration is set up or a marker variable for when the loop will be terminated

01. FOR Loop

for Loop

for is a form of looping where the block statement will be executed repeatedly according to the specified number of iterations.

For example:

```python
# For
for i in "data":
    print(i)
```

```
d
a
t
a
```

```python
# Using format method() and referring
for i in "data":
    print(f"huruf {i}")
```

```
huruf d
huruf a
huruf t
huruf a
```

for Loop

For example:

```python
# Using range function
for i in range(1,7):
    print(i)
```

```
1
2
3
4
5
6
```

```python
# Using conditional statement
for i in range(1,7):
    if i % 2 == 0 :
        print(f"{i} adalah genap")
    else:
        print(f"{i} adalah ganjil")
```

```
1 adalah ganjil
2 adalah genap
3 adalah ganjil
4 adalah genap
5 adalah ganjil
6 adalah genap
```

for Loop

For example:

```python
# Using break keyword
for i in range(1,7):
    if i % 2 == 0 :
        print(f"{i} adalah genap")
        break
```

```
2 adalah genap
```

```python
# Using continue keyword
for i in range(1,7):
    if i % 2 == 0 :
        print(f"{i} adalah genap")
        continue
```

```
2 adalah genap
4 adalah genap
6 adalah genap
```

02. WHILE Loop

while Loop

The **while** loop is an indefinite or even infinite loop. A block of code will be executed continuously as long as a condition is met. If a condition is not met in the 10th iteration, the loop will stop.

For example:

```python
# Using input function
angka = int(input("Enter number : "))

while angka< 5:
    angka += 1
    print(f"bilangan ke {angka}")
```

```
Enter number : 0
bilangan ke 1
bilangan ke 2
bilangan ke 3
bilangan ke 4
bilangan ke 5
```

while Loop

For example:

```python
# Using break keyword
angka = int(input("Enter number : "))

while angka< 20:
    angka += 2
    if angka == 10:
        break
    print(angka)
```

```
Enter number : 0
2
4
6
8
```

```python
# Using continue keyword
angka = int(input("Enter number : "))

while angka< 20:
    angka += 2
    if angka == 10:
        continue
    print(angka)
```

```
Enter number : 0
2
4
6
8
12
14
16
18
20
```

03. Break Statement

Break Statement

The break statement in Python terminates the current loop and resumes execution at the next statement, just like the traditional break found in C.

The most common use for break is when some external condition is triggered requiring a hasty exit from a loop. The break statement can be used in both while and for loops.

For example:

```
[ ] for i in "Data Science":
      if i == "e":
        break
      print(i)
```

```
D
a
t
a

S
c
i
```

Break Statement

For example:

```
for i in "I believe in yesterday":
  if i == "d":
    break
  print(i)
```

```
I

b
e
l
i
e
v
e

i
n

y
e
s
t
e
r
```

04. Continue Statement

Continue Statement

The continue statement in Python returns the control to the beginning of the while loop.

The continue statement rejects all the remaining statements in the current iteration of the loop and moves the control back to the top of the loop.

The continue statement can be used in both while and for loops.

For example:

```
[ ] for i in "Data":
      if i == "a":
        continue
      print(i)
```

```
D
t
```

Continue Statement

For example:

```
[ ] for i in "I believe in yesterday":
      if i == "a":
        continue
      elif i == "e":
        continue
      print(i)
```

```
I

b
l
i
v

i
n

y
s
t
r
d
y
```

05. Pass Statement

Pass Statement

Pass Error is the process of passing errors without stopping a program from running.

Check out the example on the right!

```python
import sys

n = " "

while(n != "exit"):
    try:
        n = (input("The Input : "))
        print(f"You get {int(n)}")
    except:
        if n == "z":
            pass
        else:
            print("The Error {}".format(sys.exc_info()[0]))
```

```
The Input : 20
You get 20
The Input : 30
You get 30
The Input : abc
The Error <class 'ValueError'>
The Input : exit
The Error <class 'ValueError'>
```

06. List Comprehension

List Comprehension

Definition:

- List Comprehension is a feature of Python lists that is used to create a new list from the elements of an existing list.

- A list comprehension is a programing language construct for creating a list based on existing lists

Syntax: [**expression** for **item** in **list**]

when using if for list comprehension we can use syntax: [**expression** for **item** in **iterable** if **condition == True**]

List Comprehension

For example:

Without using list comprehension

```
[1] #Iterating through a integer without Using List Comprehension
    number = [2,5,7]
    new_list=[]

    for i in number:
      new_list.append(i**2)
    print(new_list)

    [4, 25, 49]
```

With using list comprehension

```
[9] #Iterating through a integer Using List Comprehension
    number = [2,5,7]

    new_list = [i**2 for i in number]
    print(new_list)

    [4, 25, 49]
```

List Comprehension

For example using if function:
Without using list comprehension

```
# Using if without List Comprehension
number = range(20)
new_number=[]

for i in number:
  if i < 10:
   new_number.append(i)
print(new_number)
```

```
[0, 1, 2, 3, 4, 5, 6, 7, 8, 9]
```

With using list comprehension

```
[4]  # Using if with List Comprehension
     number = range(20)
     new_number = [i for i in number if i<10]
     print(new_number)
```

```
[0, 1, 2, 3, 4, 5, 6, 7, 8, 9]
```

List Comprehension

For example using nested if function:

Without using list comprehension

```
[5] # Using Nested IF without List Comprehension
    number = range(20)
    new_number= []

    for i in number:
      if i % 2 ==0:
        if i % 4 == 0:
          new_number.append(i)
    print(new_number)

    [0, 4, 8, 12, 16]
```

With using list comprehension

```
[6] # Using Nested IF with List Comprehension
    number = range(20)
    new_number = [i for i in number if i % 2 == 0 if i % 4 == 0]
    print(new_number)

    [0, 4, 8, 12, 16]
```

List Comprehension

For example using nested loops:

Without using list comprehension

```
[7] # nested loops without list comprehension
    number = [7,8,3,1,2]
    number2 = [9,0,7,6,3]

    new_list= []

    for i in number:
      for j in number2:
        if i == j:
          new_list.append(i)
    print(new_list)

    [7, 3]
```

With using list comprehension

```
[8] # nested loops with list comprehension
    number = [7,8,3,1,2]
    number2 = [9,0,7,6,3]

    new_number = [i for i in number for j in number2 if i == j]
    print(new_number)

    [7, 3]
```

07. Syntax Errors

Syntax Error

Syntax errors are perhaps the most common kind of complaint you get while you are still learning Python.

Syntax Error

For example:

A syntax error because a colon(:) is missing

```
#a syntax error message because after the if statement a colon: is missing.
i = 7
while i < 6
  print (i)
```

```
  File "<ipython-input-2-641a469ce09a>", line 3
    while i < 6
                ^
SyntaxError: invalid syntax
```

SEARCH STACK OVERFLOW

should have added a colon

08. Exceptions

Exceptions

An error that is detected during execution and is not unconditionally fatal, Even if a statement or expression is syntactically correct

Exceptions

For example:

A syntax error because a varibel is not difined.

the letter “a” should be replaced with the letter “A”

```
#a syntax error message because variabel non defined
A = "hello world"
print(a)
```

```
---------------------------------------------------------------------------
NameError                                 Traceback (most recent call last)
<ipython-input-3-dda01c5e4676> in <module>()
      1 #a syntax error message because variabel non defined
      2 A = "hello world"
----> 3 print(a)

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

SEARCH STACK OVERFLOW

09. Exception Handling

Exception Handling

An exception can be defined as an unusual condition in a program resulting in an interruption in the flow of the program.

Python provides a way to handle the exception so that the code can be executed without any interruption. If we don't handle the exception, the interpreter doesn't execute all the existing code after the exception.

Here, we will show how to handle exception in:

1. ZeroDivisionError
2. FileNotFound
3. KeyError

Exception Handling

ZeroDivisionError It occurs when a number is divided by a zero.

For example:

```
x = 0

division = 50/x
```

with this statement, by default, Python will return this exception

```
---------------------------------------------------------------------------
ZeroDivisionError                         Traceback (most recent call last)
<ipython-input-5-58227796e8a3> in <module>()
----> 1 division = 50/x

ZeroDivisionError: division by zero
```

SEARCH STACK OVERFLOW

Exception Handling

ZeroDivisionError

Exception handling in ZeroDivisionError is done so that the application no longer exit execution due to an error, but instead it prints a message to the screen.

For example:

with this statement, Python will print a message

```
x = 0

try:
  div = 50/x
  print(div)
except ZeroDivisionError:
  print("can't divide a number by zero")
```

```
can't divide a number by zero
```

Exception Handling

FileNotFound It occurs when a file is not found. It may be local or global.

For example:

```
[ ] open('empty.py')
```

with this statement, by default, Python will return this exception

```
---------------------------------------------------------------------------
FileNotFoundError                         Traceback (most recent call last)
<ipython-input-8-f2119fd5cece> in <module>()
----> 1 open('empty.py')

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

SEARCH STACK OVERFLOW

Exception Handling

FileNotFound

Exception handling in FileNotFound is done so that it prints a message to the screen.
For example:

with this statement, Python will print a message

```
try:
  with open('empty.py') as file:
    print(file.read())
except FileNotFoundError:
  print("Not Found 'empty.py'")
```

```
Not Found 'empty.py'
```

Exception Handling

FileNotFound

Handling a FileNotFound exception as a one-element tuple.
Don't forget when writing a one-element tuple, it must still end with a comma!

For example:

with this statement, Python will print a message

```
try:
  with open('empty.py') as file:
    print(file.read())
except (FileNotFoundError, ):
    print("Not Found empty.py")
```

```
Not Found 'empty.py'
```

Exception Handling

KeyError It occurs when you try to access a key that doesn't exist in a dictionary.

For example:

```
d = {'sum':'5.0'}

print('sum: {}'.format(d['Sum']+3))
```

with this statement, by default, Python will return this exception

```
---------------------------------------------------------------------------
KeyError                                  Traceback (most recent call last)
<ipython-input-12-f805ebcc27f1> in <module>()
      1 d = {'sum':'5.0'}
      2
----> 3 print('sum: {}'.format(d['Sum']+3))

KeyError: 'Sum'
```

SEARCH STACK OVERFLOW

Exception Handling

KeyError

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

For example:

with this statement, Python will print a message

```
d = {'sum' : '5.0'}

try:
  print('sum: {}'.format(d['Sum']+3))
except KeyError:
  print('key not found in dictionary')
except (ValueError, TypeError):
  print('Invalid value or type')
```

```
key not found in dictionary
```

Exception Handling

KeyError

In different case, for example:

```
x = {'sum': '5.0'}

try:
  print('total result: {}'.format(int(x['sum'])))
except (ValueError, TypeError) as y:
  print('handling eror: {}'.format(y))
```

with this statement, Python will print a message

```
handling eror: invalid literal for int() with base 10: '5.0'
```

10. Functions

Functions

Functions is a process that relates between an input and an output. It's also a way to organize codes for reusability. Python provides built-in functions, but we can still make our own functions.

Function name

An identifier by which the function is called

Arguments

Contains a list of values passed to the function

```
def name(arguments):
    statement
    statement
        . . .
    return value
```

Indentation

Function body must be indented

Function body

This is executed each time the function is called

Return value

Ends function call & sends data back to the program

Functions

Defining a Function

In Python, a function is defined with the use of keyword **def** followed by the name of the function and parenthesis ().

```
def greetings():
  print("Hello, world!")
```

Calling a Function

To call a function, use the function name followed by parenthesis.

```
greetings()
```

```
Hello, world!
```

Functions

Argument

While using function, an **argument** is the value that is sent to the function **when it is called**.

```
# in this function definition, x is a parameter
def add(x):
    added = x + 2
    print("{} added with 2 is {}".format(x, added))
```

Parameter

While using function, a **parameter** is the value listed inside the parentheses **in the function definition**.

```
# in this function call, 3 is an argument
add(3)
```

```
3 added with 2 is 5
```

Function

Return

A **return [*expression*]** statement is used to make program execution exit the current function state, while also returning a specified value. But we can also make the function to return nothing with **return *none***.

Has a return statement which returns a specified value

```
def add(x):
    added = x + 2
    print("{} added with 2 is {}".format(x, added))
    return added

addition = add(2)
print("Return value of add function =", addition)
```

```
2 added with 2 is 4
Return value of add function = 4
```

Doesn't have a return statement, supposed **return *none***. Considered as a procedure.

```
def add(x):
    added = x + 2
    print("{} added with 2 is {}".format(x, added))

addition = add(2)
print("Return value of add function =", addition)
```

```
2 added with 2 is 4
Return value of add function = None
```

Function

Pass by Reference

Pass by Reference means that the argument passed to the function **is a reference to a variable that already exists** in memory.

Any operation performed by the function on the variable **will be directly reflected** to the function caller.

Pass by Value

Pass by Value means that the function is provided with **a copy of the argument variable passed to it** by the caller.

So, **the original variable stays intact** and **all changes made are to a copy** of the same variable and stored at different memory locations.

Function

Pass by Reference

```python
def modify_content(b_list):
    print("Received list =", b_list)
    b_list.append(7)
    print("Modified list =", b_list)

a_list = [1, 3]

print("Before, a_list =", a_list)
modify_content(a_list)
print("After, a_list =", a_list)
```

```
Before, a_list = [1, 3]
Received list = [1, 3]
Modified list = [1, 3, 7]
After, a_list = [1, 3, 7]
```

Check out the difference!

Pass by Value

```python
def modify_content(b_list):
    print("Received list =", b_list)
    b_list = [9, 8]
    print("Modified list =", b_list)

a_list = [1, 3]

print("Before, a_list =", a_list)
modify_content(a_list)
print("After, a_list =", a_list)
```

```
Before, a_list = [1, 3]
Received list = [1, 3]
Modified list = [9, 8]
After, a_list = [1, 3]
```

Thanks!