HELLO WORLD, READY TO LEARNING BASIC PYTHON?

Looping and Error? No Problem!

By: Matplotlib Group

Looping and Error? No Problem!

MATPLOTLIB GROUP

1. Tobias Mikha Sulistiyo
2. Daud Ibadurahman
3. Putri Reghina Hilmi Prasati
4. Sari Yuliastuti
5. Fitri Alfaqrina

OUTLINES

- For Looping
- While Looping
- Break and Continue
- Pass
- List Comprehension
- Syntax Errors
- Exceptions
- Handling Error

For

For looping

For loop in Python

The for loop is used to iterate over a sequence such as a list, string, tuple, other iterable objects such as range.

```
# Using for loop on list
data = ["analyst", "engineer", "science"]

# Using for loop
# loop will run the code for each item in the list
for job in data:
    print(job)
```

```
analyst
engineer
science
```

For Loop Syntax

Use for keyword to define for loop and the iterator is defined using in the keyword.

Looping a range of number

The range() function returns a sequence of numbers starting from **0** (by default) and it increments by 1 until a final limit is reached.

For Loop Syntax

```
# Looping over string
loop = "Matplotlib"
for my_team in loop:
    print(my_team)
```

```
M
a
t
p
l
o
t
l
i
b
```

Loop a range of number

```
# Looping between a range of numbers
# Looping from 5 to 9
for number in range(5, 10): # 10 not included
    print(number, end=" ") # make output as row
    print()
```

```
5
6
7
8
9
```

Nested for loop

A nesting loop means to loop over a sequence of sequences. This is useful to loop over a sequence of items and then loop over the items in that sequence (a for loop inside another for a the loop).

```
rows = 5
# outer loop
for i in range(1, rows + 1):
    # inner loop
    for j in range(1, i + 1):
      print(j, end=" ")
    print("")
# for loop with else
else:
    print("Loop Finished")
```

```
1
1 2
1 2 3
1 2 3 4
1 2 3 4 5
Loop Finished
```

While

in looping

While

While loop is basically used to know how many iterations are required. The while loop can execute a set of statements as long as a certain condition is true. When the condition becomes false, program control passes to the line immediately following the loop.

```
number = int(input())

while (number < 20):
   number = number + 3
   print("Number is", number)
```

```
1
Number is 4
Number is 7
Number is 10
Number is 13
Number is 16
Number is 19
Number is 22
```

Break

in looping

Break

The break statement is used to terminate (stop) the loop, when a condition is met. Use the break keyword with if statement to stop and exit the loop.

```
variabel = "Data Science Track"

for job in variabel:
  if job == variabel [13]:
    # breaking a loop
    break
  print(job)
```

```
D
a
t
a

S
c
i
e
n
c
e
```

Continue

in looping

Continue

The continue statement means skipping the current iteration of a loop and continuing (immediately jumps) with the next iteration. Use the continue keyword to skip a block in a loop the current iteration.

```python
# continue loop when integer 7
for integer in range(10):
    if integer == 7:
        continue
    print(integer)
```

```
0
1
2
3
4
5
6
8
9
```

Pass

In Looping

Pass

If we want a statement or block of statements (statements), but do nothing to continue execution in order. It can also be said that the pass statement is used as a code block material, to fill an empty code block because if it is emptied, the python interpreter will give an error.

Multiple Places for Pass

1
```
for i in range(5):
  pass
```

2
```
class contoh:
  pass
```

3
```
def hitung_luas_lingkaran(diameter):
  pass
```

4
```
if 5 > 3:
  pass
else:
  pass
```

5
```
try:
  luas = 10 * '20'
except:
  pass
```

Pass

```
n = ""

while(n != "kelompok matplotlib"): #exit makes the next number unchecked
  n = (input("peroleh: "))
  print('dapat angka {}'.format(int(n)))
```

```
peroleh: 1
dapat angka 1
peroleh: kelompok matplotlib
---------------------------------------------------------------------------
ValueError                                Traceback (most recent call last)
<ipython-input-2-9353c8208636> in <module>()
      3 while(n != "kelompok matplotlib"): #exit makes the next number unchecked
      4   n = (input("peroleh: "))
----> 5   print('dapat angka {}'.format(int(n)))

ValueError: invalid literal for int() with base 10: 'kelompok matplotlib'
```

Example of data after letters, numbers cannot be checked again or value error

```
import sys
n = ''

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

```
peroleh: 1
dapat angka1
peroleh: 2
dapat angka2
peroleh: 3
dapat angka3
peroleh: 4
dapat angka4
peroleh: 5
dapat angka5
peroleh: matplotlib
```

The data can still be run or restarted even though the data has exited with a letter or other.

List Comprehension

Inline Loop and If

List Comprehension

List comprehensions are an easy way to define and create lists in Python.

List comprehensions consist of an expression followed by a for statement enclosed in brackets [ ]. Using list comprehensions we can create lists automatically in one command line. This is very useful if the list members we want to create is quite a lot.

```
angka = [2,3,4]

kubik = [] #Pangkat 3

kubik

[]

for i in angka:
  kubik.append(i**3) #menambahkan
print(kubik)

[8, 27, 64]

import math

num = [4,9,16]

kubik = []

for i in num:
  kubik.append(math.sqrt(i))
print(kubik)

[2.0, 3.0, 4.0]
```

List Comprehension

```
kelompok = ["matplotlib", "mantap"]
title = [_.title() for _ in kelompok]
print(title)
```

```
['Matplotlib', 'Mantap']
```

```
kelompok = ["matplotlib", "mantap"]

proper = []

for i in kelompok:
  proper.append(i.title())
print(proper)
```

```
['Matplotlib', 'Mantap']
```

Writing in another way
new_list = [expression
for_loop_one_or_more conditions]

```
angka = range(1, 8, 2)
kuadrat_kubik = [[_**2, _**3] for _ in angka]
print(kuadrat_kubik)
```

```
[[1, 1], [9, 27], [25, 125], [49, 343]]
```

```
for i in range(1, 8, 2):
  print(i)
```

```
1
3
5
7
```

Underscores include valid variable names in general "_" commonly used as throwaway variables (unimportant variables)

Syntax Errors and Exception

in Python

Syntax Errors ?

Syntax error is an error caused by not following the proper structure (syntax) of the language in Python. Syntax errors usually occur due to writing errors or incomplete syntax writing so that python cannot process the desired command

```
#Syntax error due to missing an indented block in the for looping

numbers = [1,2,3]

for i in numbers :
print (i)
```

```
  File "<ipython-input-2-3152b13fdc3e>", line 4
    print (i)
        ^
IndentationError: expected an indented block
```

```
#Syntax error due to missing a colon (:) in if statement

if a < 3
print("This is number three")
```

```
  File "<ipython-input-3-440e4ebdc342>", line 1
    if a < 3
           ^
SyntaxError: invalid syntax
```

Exceptions ??

Even if we have written a statement or expression from Python correctly, there is a possibility that an error will occur when the command is executed

```
#Runtime error

print(numbers)
#Error for not assigning variable before
```

```
---------------------------------------------------------------------------
NameError                                 Traceback (most recent call last)
<ipython-input-1-a2aabdc73a85> in <module>()
----> 1 print(numbers)

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

```
#Logical Error

first = '1'
first + 2
#Error due to string type variable operated with integer type
```

```
---------------------------------------------------------------------------
TypeError                                 Traceback (most recent call last)
<ipython-input-2-a63f91958523> in <module>()
      1 first = '1'
----> 2 first + 2

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

Errors that occur while the process is in progress are called exceptions, often called an error while operating (runtime error).

Some of the common built-in exceptions in Python programming along with the error that cause them are listed below:

Exceptions	Cause of Error
IndexError	Raised when the index of a sequence is out of range.
KeyError	Raised when a key is not found in a dictionary.
NameError	Raised when a variable is not found in local or global scope.
TypeError	Raised when a function or operation is applied to an object of incorrect type.
ValueError	Raised when a function gets an argument of correct type but improper value.
ZeroDivisionError	Raised when the second operand of division or modulo operation is zero.

Handling Error

in Python

why error handling is important

Because When an error occurs, or exception as we call it, Python will normally stop and generate an error message. When python is used in applications, usually when there is an error if we don't handle it, the application will close automatically

The following is an example of an error message

```
#Syntax error due to missing a colon (:) in if statement

if a < 3
print("This is number three")
```

```
  File "<ipython-input-3-440e4ebdc342>", line 1
    if a < 3
            ^
SyntaxError: invalid syntax
```

How To handle it?

There are many ways to handle errors, but this time we will only discuss try and except which are commonly used. For other types of error handles can be seen in the table below

Handle errors With:

1. The `try` block lets you test a block of code for errors.
2. The `except` block lets you handle the error.
3. The `else` block lets you execute code when there is no error.
4. The `finally` block lets you execute code, regardless of the result of the try- and except blocks.

Try and Except

To avoid this, we can use the try and except function. The 'try' function is used to run the error program. While 'except' to skip each type of error then we can add words

```
try:
        x = int('aku')
except ValueError:
        print("That was not a number.")
```

```
That was not a number.
```

```
try:
    print(number)
except NameError:
    print('not defined')
```

```
not defined
```

try:

```
syntax/program
```

except (name of error):

```
print message for eror
```

Can exceptions be combined ?

Exceptions can be combined with more than one exceptions as a tuple

```
try:
  x = int(input("Please enter a number 1: "))
  y = int(input("Please enter a number 2: "))
  z=x/y
  print(z)
except (ValueError,TypeError,ZeroDivisionError):
  print("That was no valid number")
```

```
Please enter a number 1: 0
Please enter a number 2: a
That was no valid number
```

Can exceptions be combined ?

Exceptions can be combined with more than one exceptions so that they can display the text for each condition

```
try:
  x = int(input("Please enter a number 1: "))
  y = int(input("Please enter a number 2: "))
  z=x/y
  print(z)
except ValueError:
  print("That was no valid number")
except ZeroDivisionError:
 print('Can\'t devide by zero')
```

```
Please enter a number 1: 1
Please enter a number 2: 0
Can't devide by zero
```

```
try:
  x = int(input("Please enter a number 1: "))
  y = int(input("Please enter a number 2: "))
  z=x/y
  print(z)
except ValueError:
  print("That was no valid number")
except ZeroDivisionError:
 print('Can\'t devide by zero')
```

```
Please enter a number 1: 1
Please enter a number 2: a
That was no valid number
```

Can we show the error solution ?

We can show how to solve it and the part of the error

```
e={'me':'0.5'}
try:
  print('number of me {}' .format(int(e['me'])))
except(ValueError,TypeError)as d:
  print('Handling: {}'.format(d))
```

```
Handling: invalid literal for int() with base 10: '0.5'
```

Thankyou