by. Null Science

Learning Python #4

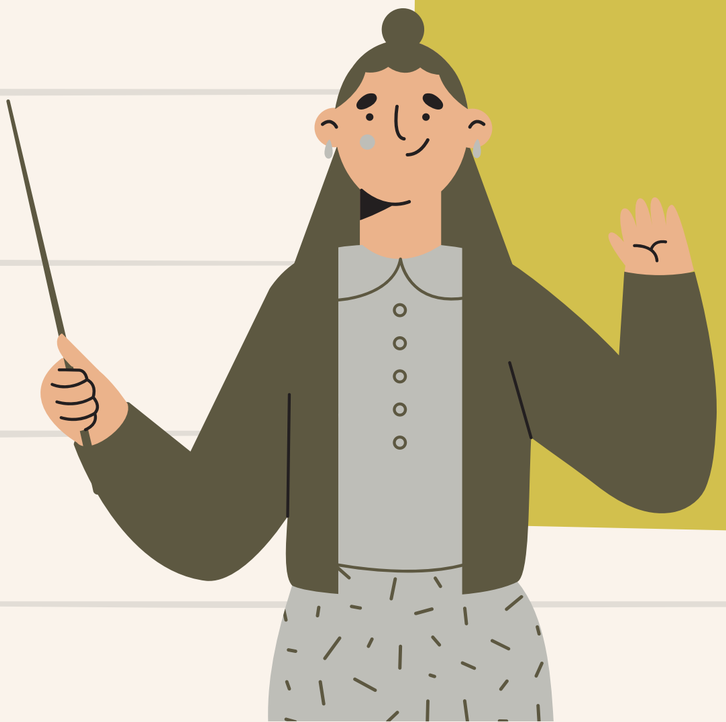
"LOOPING, HANDLING ERROR, AND HANDLING EXCEPTION"

Null Science Team

Leader : Fahmi Hamzah

Member :

- Aditya Hermawan
- Fara Rizkhi Karunia
- Nafiatul Risa
- Salsabilla Atasyaputri Setyawan



For Looping

for loop is used for iterating over a sequence (that is either a list, a tuple, a dictionary, a set, or a string).

Example

```
[3]  color = ['Red', 'Green', 'Blue']

     for i in color:
       print("Color",i)
```

```
Color Red
Color Green
Color Blue
```

Looping through String

Even strings are iterable objects, they contain a sequence of characters:

Example

```
[4]  for i in 'Looping':
       print(i)
```

```
L
o
o
p
i
n
g
```

range() function

To loop through a set of code a specified number of times, we can use the range() function.

1 Parameter

```
[16] for i in range(5):
       print(i)

     0
     1
     2
     3
     4
```

range(5) is not the values of 0 to 5, but the values 0 to 4.

2 Parameters

```
[17] for i in range(2, 10):
       print(i)

     2
     3
     4
     5
     6
     7
     8
     9
```

range(2, 10), which means values from 2 to 10 (but not including 10).

3 Parameters

```
[18] for i in range(0, 20, 3):
       print(i)

     0
     3
     6
     9
     12
     15
     18
```

specify the increment value by adding a third parameter.

FOR NESTED

A nested loop is a loop inside a loop.

```
[11] for i in range(5): #row
       for j in range(5): #column
         if j<=5:
           print("*", end=" ")
       print()
```

```
[12] for i in range(5): #Row
       for j in range(i+1): #Column
         print("*", end=" ")
       print()
```

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

OUTPUT

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

break Statement

with the break statement we can stop the loop before it has looped through all the items.

```
[8]  for i in "Data Science":
       if i==" ":
         break #Stop right here
       print(i)
```

```
D
a
t
a
```

continue Statement

with the continue statement we can stop the current iteration of the loop, and continue with the next.

```
[10] par = "This is example for Continue Statement"

     for i in par:
       if i=="a" or i=="i" or i=="u" or i=="e" or i=="o":
         continue
       print(i, end="")
       #print(i)
```

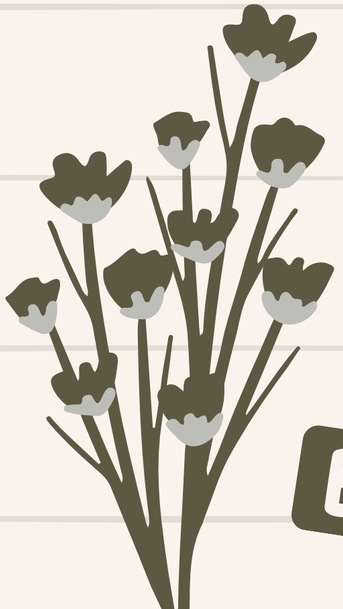
```
Ths s xmpl fr Cntn Sttmnt
```

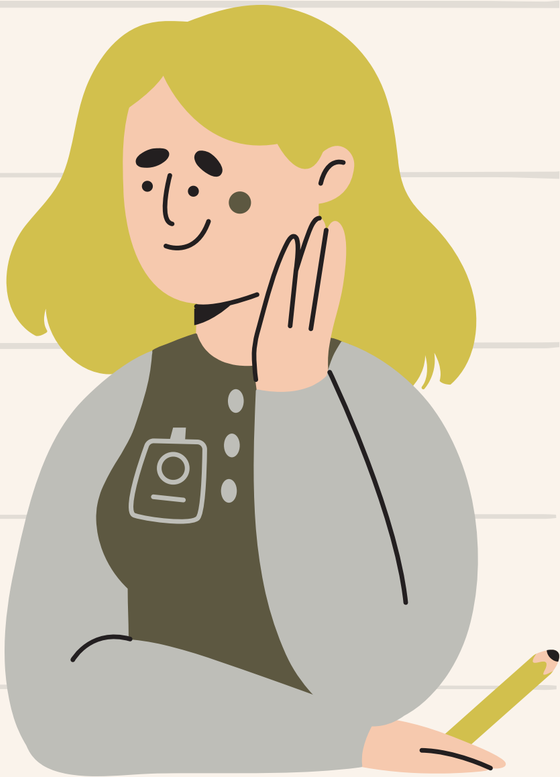
Else after For

..... usually using for searching data

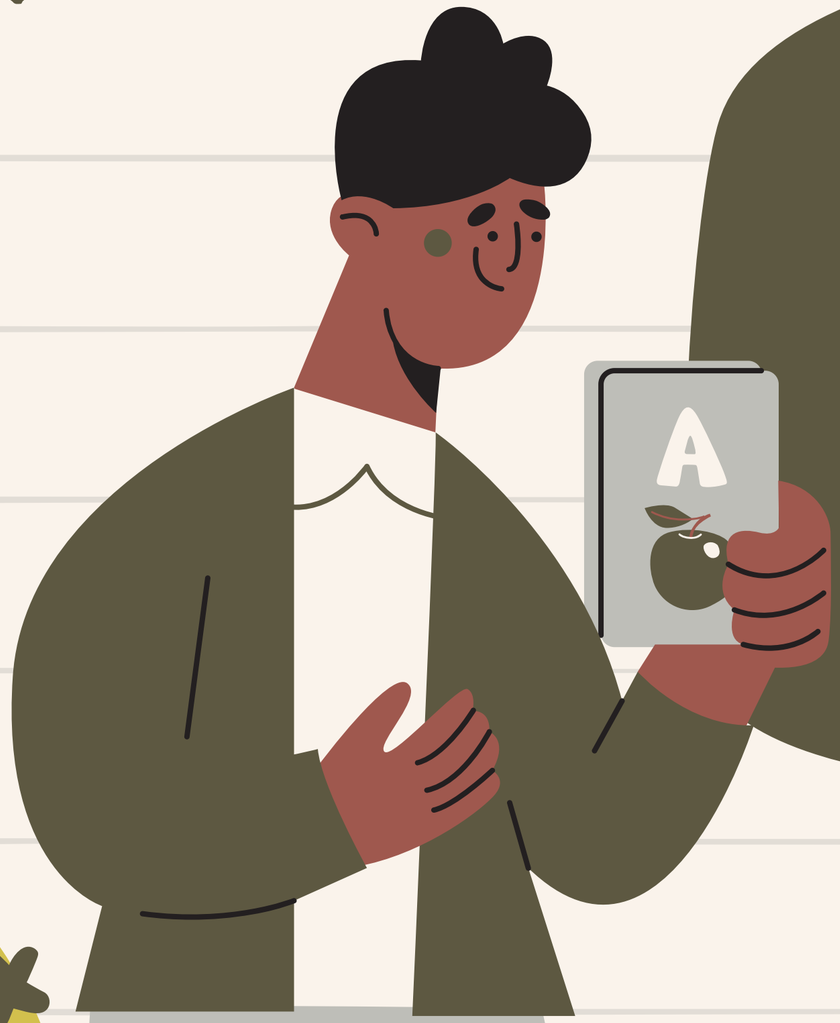
```
[20] #Find First Even Number

     for i in range(1, 7):
       if i % 2 == 0:
         print("{} is Even". format(i))
         break #Stop if True
       else:
         print("{} is Odd". format(i))
```

```
1 is Odd
2 is Even
```





While



While

While can execute a set of statements as long as a condition is true

When the condition becomes false, the line immediately after the loop in the program is executed

INPUT

```
a = int(input("Insert a Limit Number : "))
i = 3

while (i>=3 and i<=a):
  print("Number {}". format(i))
  i+=1
```

OUTPUT

```
Insert a Limit Number : 5
Number 3
Number 4
Number 5
```

Else after While

Else is only executed when a while condition becomes false

INPUT

```
a = 10

while(a<=15):
  print(a)
  a+=1
else :
  print("loop selesai")
```

OUTPUT

```
10
11
12
13
14
15
loop selesai
```

Break, Continue, and Pass Statement

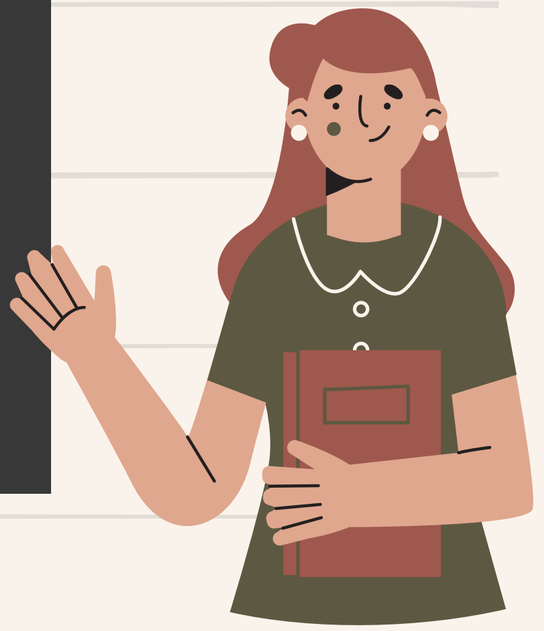
Break Statement

- The break statement in Python terminates the current loop and resumes execution at the next statement
- The break statement can be used in both while and for loops

Example

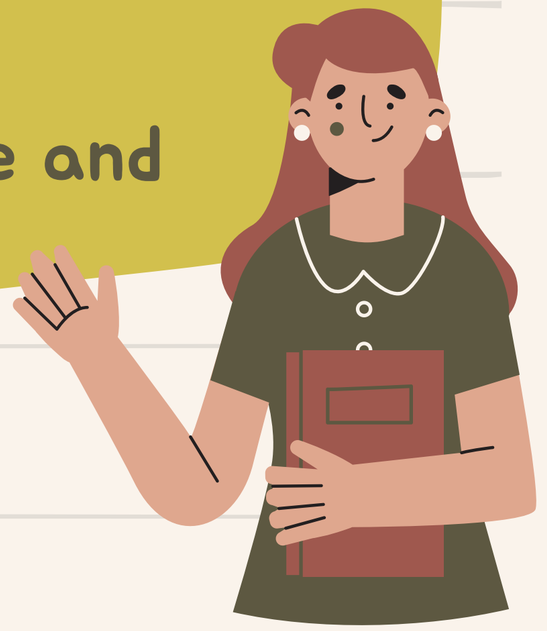
```
for i in "Learning Python from Youtube" :
  if i == "f" :
    break
  print(i)
```

```
L
e
a
r
n
i
n
g

P
y
t
h
o
n
```



Continue Statement

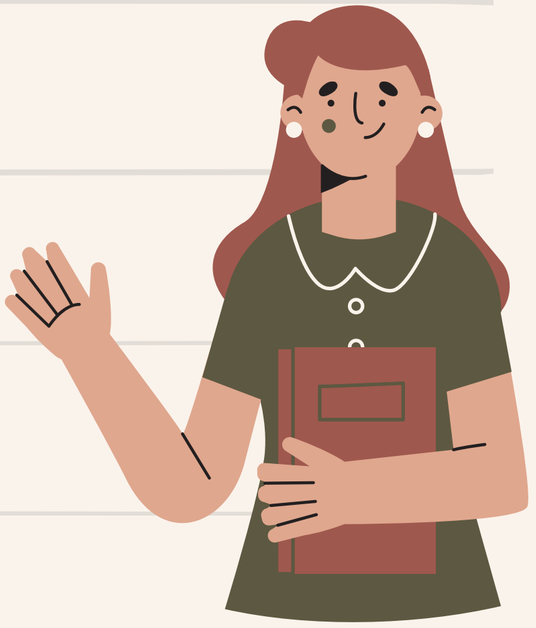
- The continue statement in Python returns the control to the beginning of the while loop
- The continue statement rejects all the remaining statements in the current iteration of the loop and moves the control back to the top of the loop
- The continue statement can be used in both while and for loops



Example

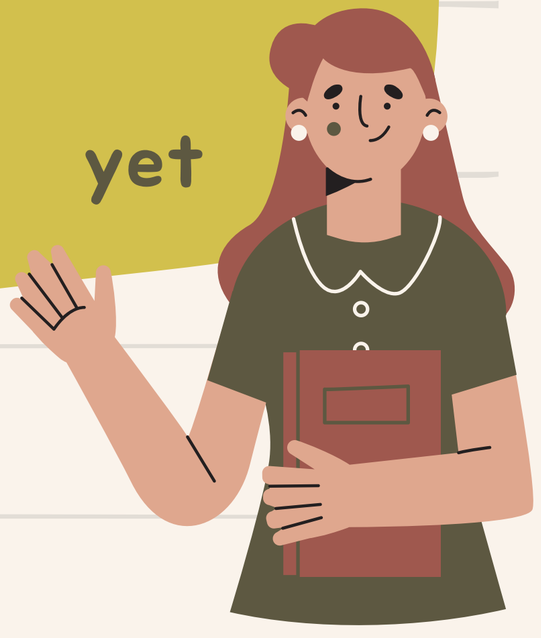
```
for i in "Learning" :
  if i == "n" :
    continue
  print(i)
```

```
L
e
a
r
i
g
```



Pass Statement

- The pass statement is used as a placeholder for future code.
- When the pass statement is executed, nothing happens, but the avoid getting an error when empty code is not allowed.
- Pass statement is used when it's not yet implemented, and keeps the program running.



Example

```
n = ""

while(n != "z"):
  n = (input("Peroleh : "))
  print("Mendapat angka {} ".format(int(n))) #should numeric
```

```
Peroleh : 4
Mendapat angka 4
Peroleh : 5
Mendapat angka 5
Peroleh : 6
Mendapat angka 6
Peroleh : g
---------------------------------------------------------------------------
ValueError                                Traceback (most recent call last)
<ipython-input-5-ed5a3ebbaae8> in <module>()
      3 while(n != "z"):
      4   n = (input("Peroleh : "))
----> 5   print("Mendapat angka {} ".format(int(n))) #should numeric

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

```
6import sys #error checking (try and except)

n = ''

while(n != "exit"):
  try:
    n = input("Enter : ")
    #n = int(input("Enter : "))
    print("Angka {} ".format(int(n))) #want integer
  except:
    if n=="exit":
      pass #will stop
    else:
      print("Error {} ".format(sys.exc_info()[0]))
```

```
Enter : 4
Angka 4
Enter : 5
Angka 5
Enter :
Error <class 'ValueError'>
Enter : g
Error <class 'ValueError'>
Enter :
```

List Comprehension

List comprehension in Python is an easy and compact syntax for creating a list from a string or another list. It is a very concise way to create a new list by performing an operation on each item in the existing list. List comprehension is considerably faster than processing a list using the for loop.

List Comprehension

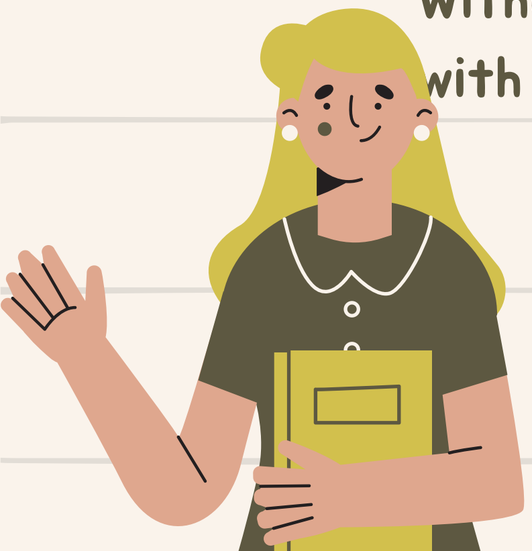
List Comprehension

List comprehension offers a shorter syntax when you want to create a new list based on the values of an existing list.

Example:

Based on a list of groceries, you want a new list, containing only the groceries with the letter "e" in the name.

Without list comprehension you will have to write a for statement with a conditional test inside:



Example 1:

```
[1] groceries = ["broom", "soap", "rice", "bread", "cereal", "fork", "spoon", "eyeliner"]
    newlist = []

    for x in groceries:
      if "e" in x:
        newlist.append(x)

    print(newlist)
```

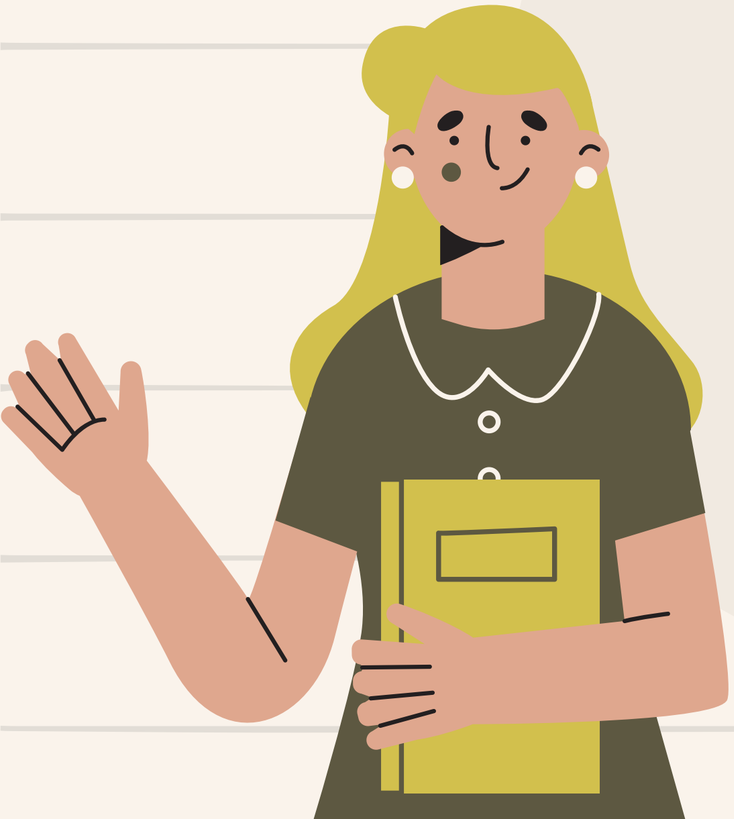
```
['rice', 'bread', 'cereal', 'eyeliner']
```

Example 2:

Or in the list a number, you want a new list, that can perform mathematical operations on ii

```
num1 = range(1, 20)
num2 = range(1, 20)

newlist = []

for x in num1:
  for y in num2:
    if x//y == 4 and x%y == 2:
      newlist.append([x, y])
print(newlist)
```

```
[[14, 3], [18, 4]]
```



With list comprehension you can do all that with only one line of code:
The syntax:
newlist = [expression for item in iterable if condition == True]

The return value is a new list, leaving the old list unchanged.

```
[8]  groceries = ["broom", "soap", "rice", "bread", "cereal", "fork", "spoon", "eyeliner"]
     newlist = [x for x in groceries if "e" in x]
     print(newlist)

     ['rice', 'bread', 'cereal', 'eyeliner']
```

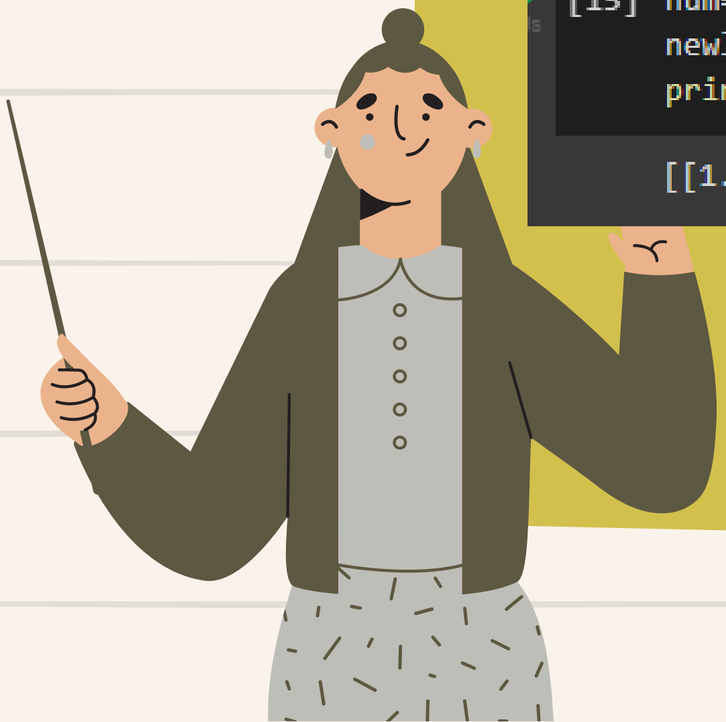
```
[9]  num1 = range(1, 20)
     num2 = range(1, 20)

     newlist = [x for x in num1 for y in num2 if x//y == 4 and x%y == 2]
     print(newlist)

     [14, 18]
```

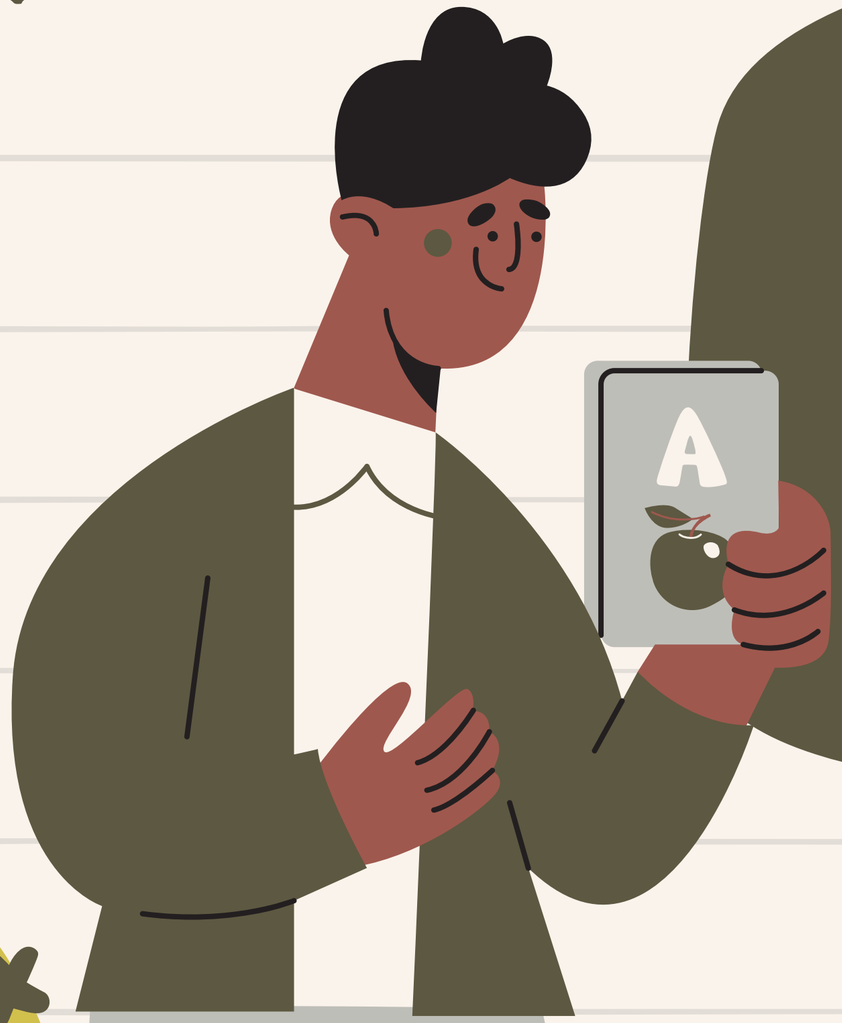
Source Code and Output

Another list comprehension

- Underscores include valid variable names
- In general "_" is used as a throwaway variable (unimportant variable)

```
[15] num=range(1,8,2)
     newlist = [[_**0.5, _**2] for _ in num]
     print(newlist)

     [[1.0, 1], [1.7320508075688772, 9], [2.23606797749979, 25], [2.6457513110645907, 49]]
```





Syntax Error

"Syntax errors or often called parsing errors."

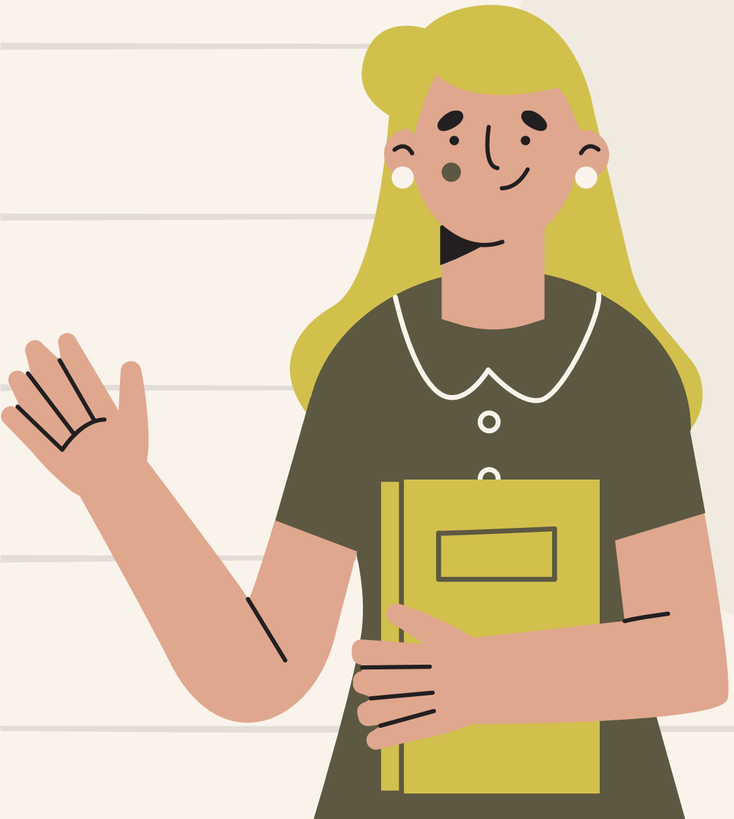
Syntax errors occur when Python can't understand what you're commanding. Whereas exceptions (errors while operating) occur when Python understands what you are commanding but gets problems following what you command (occurs when the application is already running).

Syntax Errors

Example of a syntax error :

- Incorrect placement of indents (spaces at the beginning).
- After the condition of the while command requires a colon (:).

Incorrect placement of indents (spaces at the beginning).



```
print('Hello World')
 File "<stdin>", line 1
  print('Hello World')

  File "<ipython-input-5-0ad6b35b9c3d>", line 2
    File "<stdin>", line 1
    ^
IndentationError: unexpected indent

SEARCH STACK OVERFLOW
```

Source Code and Output

After the condition of the while command requires a colon (:)

```
while True print('Hello world')
 File "<stdin>", line 1
  while True print('Hello world')

  File "<ipython-input-6-dca2bd7aed5e>", line 1
    while True print('Hello world')
                    ^
SyntaxError: invalid syntax

SEARCH STACK OVERFLOW
```

Source Code and Output

In a syntax error, the line where the error was detected is reappeared, followed by an arrow indicating the earliest point of the error.

The two examples above have different groups (types) of errors, the first is Indentation Error and the second is Syntax Error. Then after mentioning it, there is a detailed error message (description), such as an unexpected indent.

If using script call mode, the script file name and line number where the error occurred will be returned. As for the interactive mode in the two examples above, the filename appears as "stdin". Here's an example of calling a script named example_error_syntax.py where an error occurs on line 2.

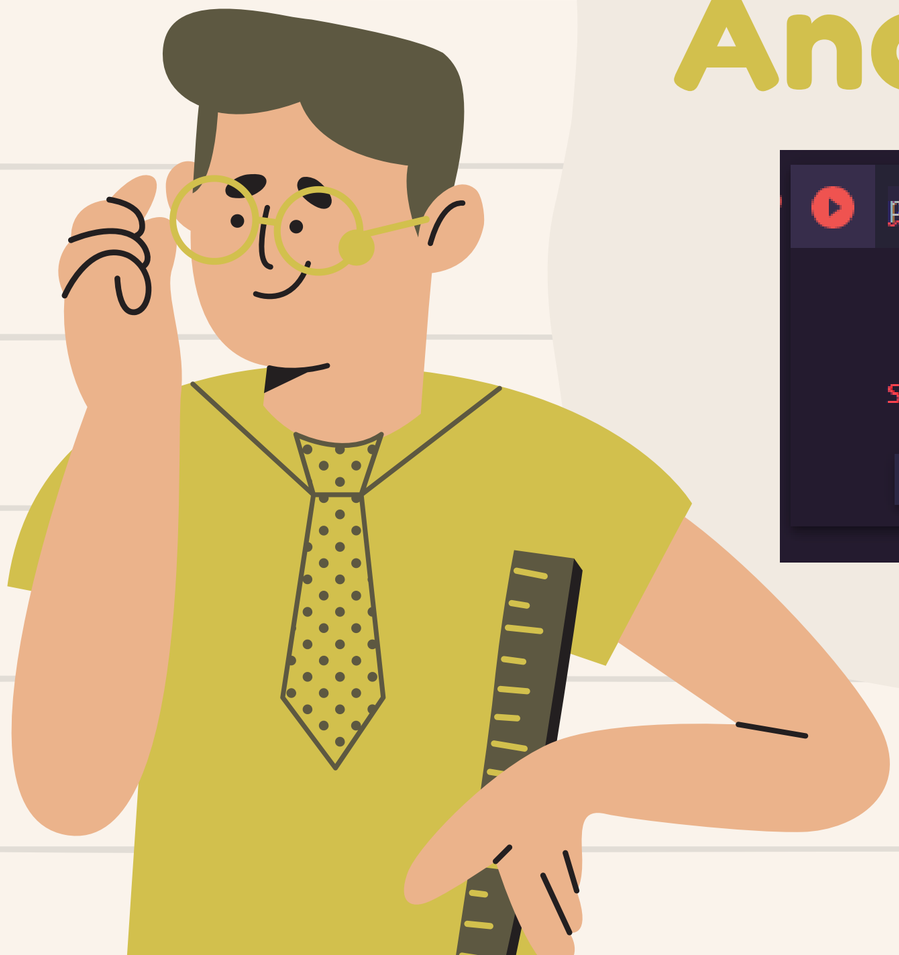
```
python syntax_error.py
 File "syntax_error.py", line 2
  if True print('Syntax is wrong')

  File "<ipython-input-7-deb18568d569>", line 1
    python syntax_error.py
                         ^
SyntaxError: invalid syntax

SEARCH STACK OVERFLOW
```

Source Code and Output

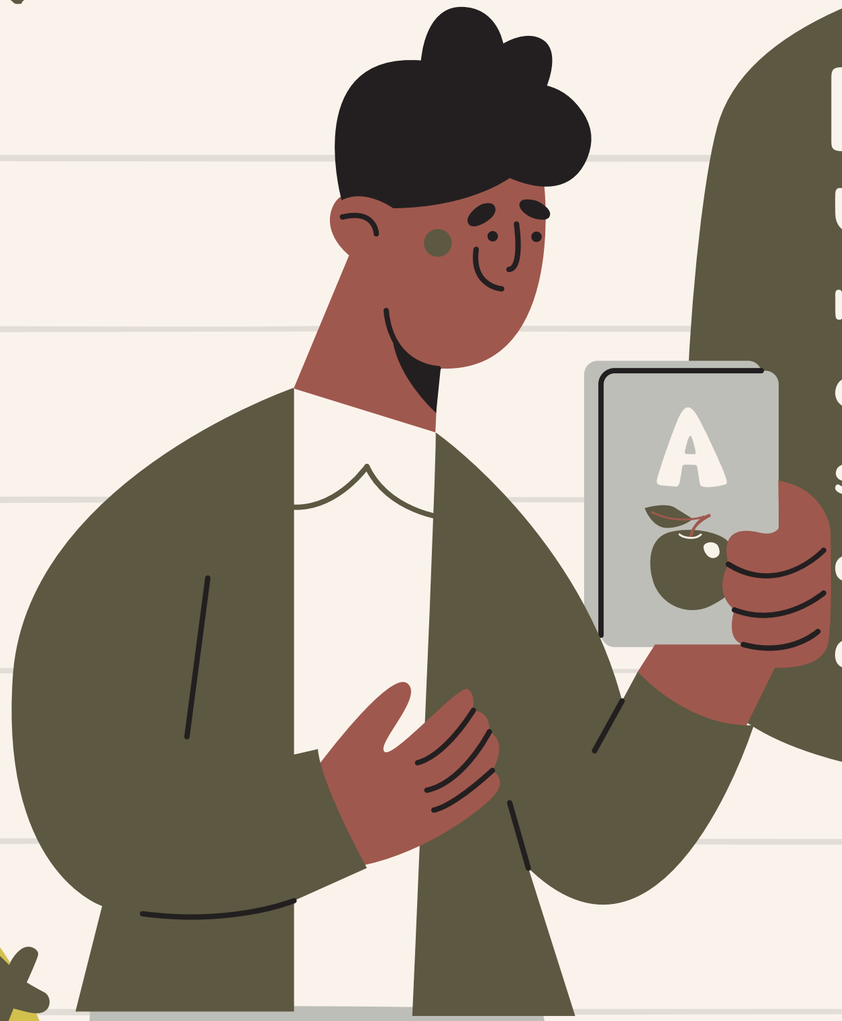
Another Example :

```
print"Hello World

  File "<ipython-input-8-ffd5c9873b15>", line 1
    print"Hello World
                      ^
SyntaxError: EOL while scanning string literal

SEARCH STACK OVERFLOW
```

Source Code and Output



Errors and Exceptions



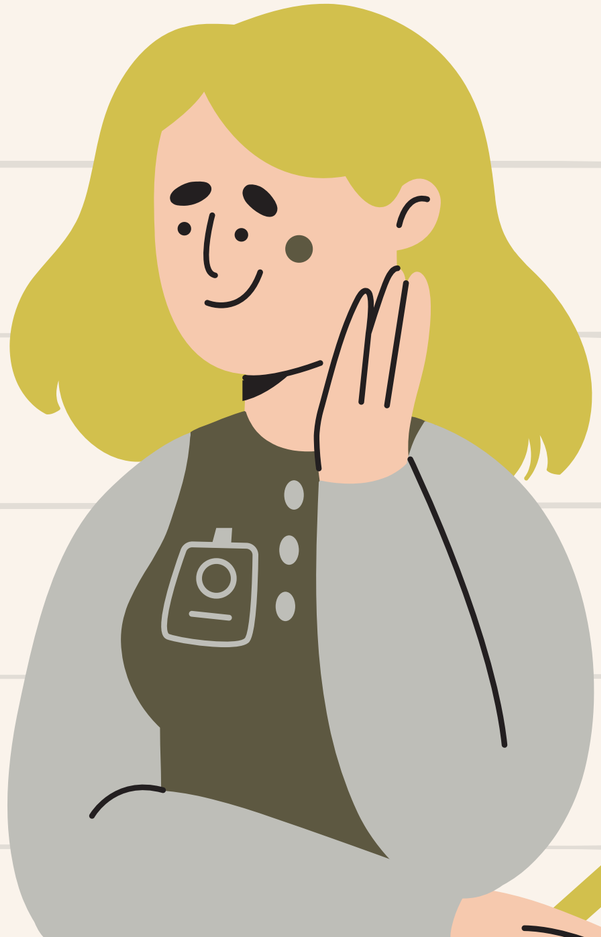
Errors

Until now error messages haven't been more than mentioned, but if you have tried out the examples you have probably seen some. There are (at least) two distinguishable kinds of errors: syntax errors and exceptions.

Syntax Errors

Syntax errors, also known as parsing errors, are perhaps the most common kind of complaint you get while you are still learning Python:

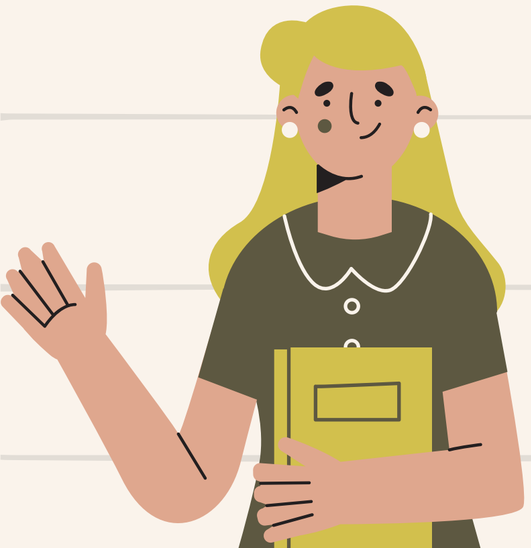
```
In [1]: while True print('Hello world')
          File "<stdin>", line 1
            while True print('Hello world')
                       ^
        SyntaxError: invalid syntax


          File "<ipython-input-1-b940826e3ab3>", line 1
            while True print('Hello world')
                       ^
        SyntaxError: invalid syntax
```



The error is caused by (or at least detected at) the token preceding the arrow: in the example, the error is detected at the function print(), since a colon (':') is missing before it.

Exceptions

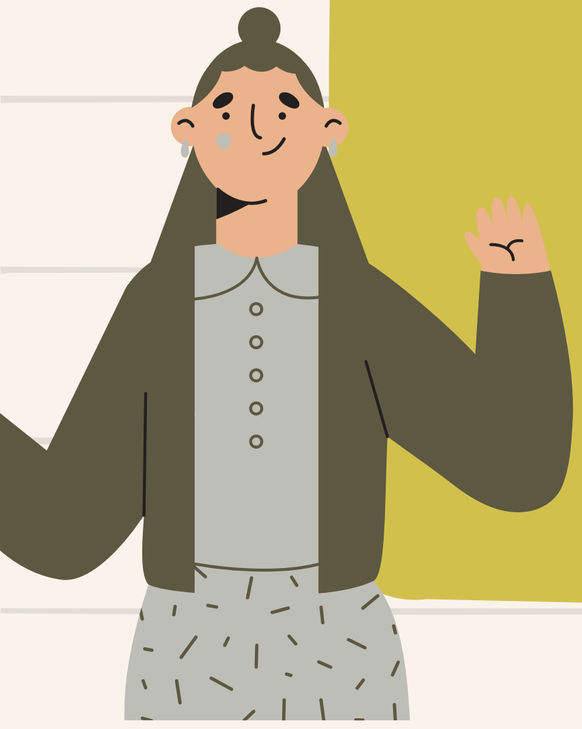
if a statement or expression is syntactically correct, it may cause an error when an attempt is made to execute it. Errors detected during execution are called exceptions and are not unconditionally fatal



```
In [6]: 10 * (1/0)

---------------------------------------------------------------------------
ZeroDivisionError                         Traceback (most recent call last)
<ipython-input-6-353959f60a93> in <module>
----> 1 10 * (1/0)
      2 4 + spam*3

ZeroDivisionError: division by zero
```

```
[5]: 4 + spam*3

---------------------------------------------------------------------------
NameError                                 Traceback (most recent call last)
<ipython-input-3-c98bb92cdcac> in <module>
----> 1 4 + spam*3

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

```
---------------------------------------------------------------------------
TypeError                                 Traceback (most recent call last)
<ipython-input-4-d2b23a1db757> in <module>
----> 1 '2' + 2

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



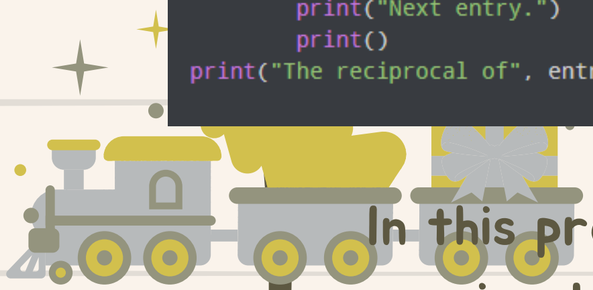
The last line of the error message indicates what happened. Exceptions come in different types, and the type is printed as part of the message: the types in the example are ZeroDivisionError, NameError and TypeError

Input

```
# import module sys to get the type of exception
import sys

randomList = ['a', 0, 2]

for entry in randomList:
    try:
        print("The entry is", entry)
        r = 1/int(entry)
        break
    except:
        print("Oops!", sys.exc_info()[0], "occurred.")
        print("Next entry.")
        print()
print("The reciprocal of", entry, "is", r)
```

Output

```
The entry is a
Oops! <class 'ValueError'> occurred.
Next entry.

The entry is 0
Oops! <class 'ZeroDivisionError'> occured.
Next entry.

The entry is 2
The reciprocal of 2 is 0.5
```



In this program, we loop through the values of the randomList list. As previously mentioned, the portion that can cause an exception is placed inside the try block.

If no exception occurs, the except block is skipped and normal flow continues(for last value). But if any exception occurs, it is caught by the except block (first and second values).

Since every exception in Python inherits from the base Exception class, we can also perform the above task in the following way:

```
# import module sys to get the type of exception
import sys

randomList = ['a', 0, 2]

for entry in randomList:
    try:
        print("The entry is", entry)
        r = 1/int(entry)
        break
    except Exception as e:
        print("Oops!", e.__class__, "occurred.")
        print("Next entry.")
        print()
print("The reciprocal of", entry, "is", r)
```

This program has the same output as the above program.