# Common mistakes and errors in python programming #5229

In this article we will cover few of the most common mistakes that one can make while programming in python. Discussing these errors could be really helpful as a lot of the times getting continuous errors while trying to solve a problem is very \*\*frustrating\*\* especially for newbie programmers and all the programmers agree upon this.

# Errors/mistakes covered are:

* EOF/NZEC error (found when using online coding platform compilers like codechef, hackerrank)
* Misusing expressions as defaults for function arguments
* Using class variables incorrectly
* Misunderstanding Python scope rules
* Modifying a list while iterating over it
* Name clashing with Python Standard Library modules
* Failing to address differences between Python 2 and Python 3
* Misusing the \_\_del\_\_ method
* '#' does not make the program ignore everything written ahead of the hashtag
* Mistakes in Indentation

# 1.) EOF/NZEC error (found when using online coding platform compilers like codechef, hackerrank):

EOFError is raised when one of the built-in functions input() or raw\_input() hits an end-of-file condition (EOF) without reading any data. This error is sometimes experienced while using online IDEs. This occurs when we have asked the user for input but have not provided any input in the input box. We can overcome this issue by using try and except keywords in Python. This is called as Exception Handling.

```python

# instead of writing this:

n = int(input())

print(n \* 10)

# We’ll write:

try:

n = int(input())

print(n \* 10)

except EOFError as e:

print(e)

```

# 2.) Misusing expressions as defaults for function arguments:

In python we can specify that a function argument is \*\*optional\*\* by giving a \*\*default value\*\* for it. But if not used carefully can lead to some confusion when the default value is \*\*mutable\*\*, eg.,

```python

# Say we have function as below

def music(type=[]): #type is optional and defaults to [] if not specified

type.append("bass") #but this line could cause problems as we'll see

return type

```

Here the common mistake will be to think that the optional argument will be set to the specified default expression each time the function is called without supplying for the optional argument.

In the above code for example, one might expect that calling ‘\*\*music()\*\*’ again and again , without specifying \*\*type\*\* argument would always return ‘bass’ , since the assumption would be that each time music() is called (without a \*\*type\*\* argument specified) \*\*type\*\* is set to [] (an empty new list).

But this isn’t what actually happens,

Lets take a look on program execution:

```python

# >>> music()

[“bass”]

# >>> music()

[“bass”, “bass”]

# >>> music()

[“bass” , “bass” , “bass”]

```

Now you must be thinking why we got the above outcome, why did it keep appending the default value (‘bass’) to an existing list each time \*\*music()\*\* was called, rather than creating a new list each time ??

The more advanced Python programming answer is that \*\*the default value for a function argument is only evaluated once, at the time that the function is defined.\*\* Thus, the \*\*type\*\* argument is initialized to its default (i.e., an empty list) only when music() is first defined, but then calls to music() (i.e., without a \*\*type\*\* argument specified) will continue to use the same list to which \*\*type\*\* was originally initialized.

The workaround for the above error is:

```python

#>>> def music(type=None):

if type is None: # or if not type:

type = []

type.append("bass")

return type

# >>> music()

["bass"]

# >>> music()

["bass"]

# >>> music()

["bass"]

```

# 3.) Using class variables incorrectly:

Lets understand this straight up with an example :

```python

class A(object):

x = 1

class B(A):

pass

class C(A):

pass

print(A.x, B.x, C.x)

#Output: 1 1 1

```

Now,

```python

B.x = 2

print(A.x, B.x, C.x)

#Ouput: 1 2 1

```

As expected

```python

A.x = 3

print(A.x, B.x, C.x)

#Ouput: 3 2 3

```

Huh ?? why did C.x change too ?? we only changed A.x??

In Python, class variables are internally handled as dictionaries and follow what is often referred to as Method Resolution Order (MRO).

So in the above code, since the attribute x is not found in class C, it will be looked up in its base classes (only A in the above example, although Python supports multiple inheritances). In other words, C doesn’t have its own x property, independent of A. Thus, references to C.x are in fact references to A.x. This causes a Python problem unless it’s handled properly.

# 4.) Misunderstanding Python scope rules:

Python scope resolution is based on **L**ocal, **E**nclosing, **G**lobal, **B**uilt-in (LEGB).

Now what the heck was that!!

This is a bit of advanced python programming so don’t worry if you were not able to understand this now.

Consider the following:

```python

x = 10

def plus():

x += 1

print(x)

plus()

#output : Traceback (most recent call last):

# File "<stdin>", line 1, in <module>

# File "<stdin>", line 2, in plus

#UnboundLocalError: local variable 'x' referenced before assignment

```

Why are we getting an error ??

The above error occurs because, when you make an assignment to a variable in a scope, that variable is automatically considered by Python to be local to that scope and shadows any similarly named variable in any outer scope.

Many are thereby surprised to get an \*\*UnboundLocalError\*\* in previously working code when it is modified by adding an assignment statement somewhere in the body of a function. (You can read more about this here.)

It is particularly common for this to trip up developers when using lists.

Consider the following example:

```python

one = [1, 2, 3]

def plus1():

one.append(5) # This works ok...

plus1()

one

[1, 2, 3, 5]

one = [1, 2, 3]

def plus2():

one += [5] #this gets error

plus2()

#Output: Traceback (most recent call last):

# File "<stdin>", line 1, in <module>

# File "<stdin>", line 2, in plus

#UnboundLocalError: local variable 'one' referenced before assignment

```

Now Why did plus2 get error while plus1 ran fine?

The answer is the same as in the prior example problem but is admittedly more subtle. plus1 is not making an assignment to \*\*one\*\*, whereas plus2 is. Remembering that one += [5] is really just shorthand for one = one + [5], we see that we are attempting to assign a value to one (therefore presumed by Python to be in the local scope). However, the value we are looking to assign to one is based on one itself (again, now presumed to be in the local scope), which has not yet been defined.

# 5.) Modifying a list while iterating over it:

example:

```python

odd = lambda x : bool(x % 2)

numbers = [n for n in range(10)]

for i in range(len(numbers)):

if odd(numbers[i]):

del numbers[i] # Not good \*\*: (\*\* : Deleting item from a list while iterating over it

#Output: Traceback (most recent call last):

File "<stdin>", line 2, in <module>

IndexError: list index out of range

```

Deleting an item from a list or array while iterating over it is a Python problem that is well known to any experienced software developer. But while the example above may be fairly obvious, even advanced developers can be unintentionally bitten by this in code that is much more complex.

Python incorporates a few elegant programming paradigms which, when used properly, can result in significantly simplified and streamlined code. A side benefit of this is that simpler code is less likely to be bitten by the accidental-deletion-of-a-list-item-while-iterating-over-it bug. One such paradigm is that of list comprehensions. Moreover, list comprehensions are particularly useful for avoiding this specific problem, as shown by this alternate implementation of the above code which works perfectly:

```python

odd = lambda x : bool(x % 2)

numbers = [n for n in range(10)]

numbers[:] = [n for n in numbers if not odd(n)]

numbers

[0, 2, 4, 6, 8]

```

# 6.) Name clashing with Python Standard Library modules:

One of the greatest appealing points in Python is the wealth of library modules to carry out various commands and tasks. But as a result, it’s not that difficult to run into a name clash between the name of one of your modules and a module in the standard library that ships with Python (for example, you might have a module named email.py in your code, which would be in conflict with the standard library module of the same name).

This can lead to problems, such as importing another library which in turns tries to import the Python Standard Library version of a module but, since you have a module with the same name, the other package mistakenly imports your version instead of the one within the Python Standard Library. This is where bad Python errors happen.

Care should, therefore, be exercised to avoid using the same names as those in the Python Standard Library modules.

# 7.) Misusing the \_\_del\_\_ method:

In Python, \_\_del\_\_ is a dunder or magic method. Magic methods are ones with two prefixes and suffix underscores in the method name. It is usually used for operator overloading.

\_\_del\_\_ is a destructor method and is called when all references of an object are deleted. That is, when an object is garbage collected.

Syntax to use \_\_del\_\_:

def\_\_del\_\_(self):

Body of destructor

# 8.) '#' does not make the program ignore everything written ahead of the hashtag:

Sometimes while solving problems online you may end up copying the code and solve first on your offline IDE instead the online version but a lot of the times theses programs contain somethin like this commented using #

```python

#!/bin/python3

```

This instead of commenting the thing in front is instead telling the program where to access the python on your system and mostly in an offline IDE like VS code you will receive an error because a path to Python has already been

```python

/bin/python3 : The term '/bin/python3' is not recognized as the name of a cmdlet, function, script file, or operable program. Check the spelling of the name, or if a

path was included, verify that the path is correct and try again.

At line:1 char:1

+ /bin/python3 "c:\PYTHON\program.py"

+ ~~~~~~~~~~~~

+ CategoryInfo : ObjectNotFound: (/bin/python3:String) [], CommandNotFoundException

+ FullyQualifiedErrorId : CommandNotFoundException

```

specified on your offline system at the time of installation , and the location specified by the ‘#’ is where the program will look for python and at this location the program won’t find any.

# 9.) Mistakes in Indentation:

Python does not rely on brackets to separate code blocks. Thus, even one indentation mistake can result in unexpected output. This becomes all the more challenging when one is using multiple functions– all in the same code file. Oftentimes, it may not even give an indentation error but can become a serious bug in the code.