10 Python Mistakes That Tell You're a Nooby

#1. Using import *

When we start learning Python, we get some bad coding habits that we're unaware of.

You might write code that gets the job done once but could fail in the future, or you might use workarounds when there's a built-in function that can make your life easier.

Most of us still have more than one of those bad Python habits that remain from our first months of learning. The good news is that you can easily fix them today with the snippets below.

1. Using import *

Whenever we feel lazy, it's tempting to import everything from a module using the `from xyz import *`.

This isn't a good practice for many reasons. To name a few:

1. It can be inefficient: If the module has a good number of objects, you'll need to wait a long time until everything is imported
2. It can cause conflict between variable names: When you use the `*` you don't know which objects you're importing as well as their names.

How to deal with this? Either import the specific object you plan to use or the whole module.

2. Try/Except: Not specifying the exception in the "except" clause

I've been neglecting this one for a long time.

I can't count how many times Pycharm let me know (with those ugly underlines) that I shouldn't use a bare `except`. This isn't recommended in the PEP 8 guidelines.

The problem with a bare `except` is that it will catch SystemExit and KeyboardInterrupt exceptions, making it harder to interrupt a program with Control-C.

Next time you use try/except, specify the exception in the `except` clause.

3. Not using Numpy for math computations

Often we forget that there are packages that can make our life easier and more productive in Python.

One of those packages you should use for math computations is Numpy. Numpy could help you solve math operations faster than `for` loops.

Say we have an array of `random_scores` and we want to get the average score of those who failed the exam (score<70). Let's try to solve this with a `for` loop.

Now let's solve this with Numpy.

If you run both, you'll see that Numpy is faster. Why? Because Numpy vectorizes our operations.

4. Not closing a file previously opened

A good practice everyone knows is that every file we open with Python must be closed.

This is why we use the `open` , `write/read`, `close` whenever we work with files. That's OK, but if the `write/read` methods throw an exception, the file won't be closed.

To avoid this issue, we have to use the `with` statement. This will close the file even if there's an exception.

5. Not following PEP8

PEP8 is a document everyone learning Python should read. It provides guidelines and best practices on how to write Python code (some of the advice in this article comes from PEP8)

This guideline might be intimidating for those new to Python. Fortunately, some PEP8 rules are incorporated in IDEs (this is how I knew about the bare `except` rule).

Say you're using Pycharm. If you write code that doesn't follow the PEP8 guidelines,

If you hover over the underlines, you'll see instructions on how to fix them.

In my case, I only have to add a white space after `,` and `:`

I also changed the name of my variable `x` to `my_name`. This isn't suggested by Pycharm, but PEP8 recommends using variable names that are easy to understand.

6. Not using .keys and .values methods properly when working with dictionaries

I think most of you know what the .keys and .values methods do when working with dictionaries.

In case you don't know, let's have a look.

```
dict_countries = {'USA': 329.5, 'UK': 67.2, 'Canada': 38}>>>dict_countries.keys()
dict_keys(['USA', 'UK', 'Canada'])>>>dict_countries.values()
dict_values([329.5, 67.2, 38])
```

The problem here is that we sometimes don't use them properly.

Say we want to loop through the dictionary and obtain the keys. You might use the .keys method, but did you know you could obtain the keys just by looping through the dictionary? In this case, using .keys will be unnecessary.

Also, we might come up with workarounds to get the values of a dictionary, but that could be easily obtained with the .items() method.

7. Never using comprehensions (or using them all the time)

Comprehension offers a shorter syntax when you want to create a new sequence (list, dictionary, etc) based on a sequence that is already defined.

Say we want to lowercase all the elements in our countries list.

Although you could do this with a for loop, you could simplify things with a list comprehension.

Comprehensions are very useful, but don't overuse them! Remember the Zen of Python: "Simple is better than complex".

8. Using range(len())

One of the first functions we learned as beginners are the `range` and `len`, so no wonder why most people have the bad habit to write `range(len())` when looping through lists.

Say we have a `countries` and `populations` lists. If we want to iterate through both lists at the same time, you'd probably use `range(len())`.

Although that gets the job done, you could simplify things using `enumerate` (or even better, use the `zip` function to pair elements from both lists)

9. Formatting with the + operator

Probably one of the first things we learn in Python is how to join strings with the `+` operator.

This is a useful, yet inefficient way to join strings in Python. Besides, it's not so good-looking — the more strings you need to join, the more + you'll use.

Instead of using this operator, you can use the f-string.

The best part about f-strings is that it's not only useful for concatenation but has different applications.

10. Using default mutable values

If you include a mutable value (like a list) as a default parameter of a function, you'd see some unexpected behavior.

```
# Bad
def my_function(i, my_list=[]):
   my_list.append(i)
   return my_list>>> my_function(1)
[1]
>>> my_function(2)
[1, 2]
>>> my_function(3)
[1, 2, 3]
```

In the code above, every time we call the `my_function` function, the list `my_list` keeps saving the values from previous calls (most likely we want to initiate an empty list every time we call the function)

To avoid this behavior, we should set this `my_list` parameter equal to `None` and include the `if` clause below.

```
# Good
def my_function(i, my_list=None):
   if my_list is None:
      my_list = []
   my_list.append(i)
   return my_list>>> my_function(1)
[1]
>>> my_function(2)
[2]
>>> my_function(3)
[3]
```