Programming Best Practices: Writing Better Code

§

Writing code can be as simple as importing the required libraries, declaring our variables, functions, and classes as required, including some docstrings here and there, some additional comments, executing, and we're done. While we're at it, let's skip the function & class part and drop everything as is. Even better, let's also save some lines by stripping our file from all comments.

The result? A piece of code with absolutely no indication of what it does or how it does it, with the impossibility of modularizing & scaling in any meaningful way. In short, a beautiful, useless creation made by us, just for us.

In this section, we will review some simple methods that will immediately bring clarity & credibility to our code, being thus able to share it with others, understand it ourselves when re-reading it some weeks or months later, and not perish in the process.

We will use Python as an example, but most steps will apply to any programming language.

We'll be using Python scripts which can be found in the Blog Article Repo.

§

Table of Contents

§

Legibility

If we were to choose one step to try and write better code, this would be the one. We can write code without any functions whatsoever or even deliver underperforming software. Still, if the people reading our code don't even know what it's for, what it's supposed to do, where to look, and what to change to optimize its performance, they're better off writing the code by themselves.

Legibility is not just about filling our script with comments on every step we do (*maybe when we're trying to explicitly teach something, it might be a good idea*). Legibility is about making our code understandable to ourselves and anyone with little to no context of what we're trying to achieve.

The good thing is, if we are to improve legibility, we have some simple and handy mechanisms additional to commenting that we can use.

1. Authoring

This is the first step we will cover and a crucial one when sharing our material. It provides the creator with a way to assume authorship of the creation. It also gives readers and consumers a way to contact the creator in case of any questions. We can get our hands on a beautifully written and extremely useful program, but who the heck does this code belong to? Was it written by ChatGPT?

It is also relevant to include authoring and contact information as metadata in the code we write, especially in a working environment, so that people will know who to blame when things go wrong (*just kidding, sort of*). Jokes aside, this is true and happens more often than not. We want to provide a way for other collaborators to reach out if our code doesn't work as expected. After all, we're assuming responsibility when submitting code to other people.

As with several commenting techniques, Authoring comes down to personal preference, though we should include at least the creation date, our name, and some contact information.

Typically, we enclose this metadata at the top of our script in a docstring using single `'` or double `"` triple quotes:

Code

```
'''
Created on: Wed Jan 18 20:48:18 2023
@author: Lucifer Morningstar
contact: lucifermorningstar@gmail.com
'''
```

We can complement this information by specifying additional fields, such as last modification date, Python version used, and more.

The nice thing about authoring is that several IDEs have the option to customize & introduce these lines by default when creating a new blank document.

2. Comments

2.1 Simple commenting

Commenting is always encouraged when working with code. It lets us re-read and understand the procedures we used some time ago and gives context to other people reading our material.

We can comment on a single line by using the hash symbol `#`:

Code

```
# This is a comment.
```

2.2 Over-commenting

The caveat of commenting is that it is often abused to the point of replicating line by line what the code is doing. Again, this is fine if we're teaching someone, but to the experienced programmer, this will not be necessary and will, in turn, introduce too many reading breaks, which are not required. We can think of over-commenting as abusing punctuation; it breaks the flow and impedes clear reading.

Fortunately, there are two simple rules to avoid over-commenting:

- Use comments to explain the why, not the how.
- Only write a comment when it's indispensable.

Both statements are important because sometimes, the code is self-explanatory. This would make writing the following comment unnecessary:

Code

```
# Declare a while loop that prints 'Hello', and stops after 8 iterations
i = 0
while i < 8:
    print('Hello')
    i += 1
```

2.3 Breaking comments in a new line

Comments are meant to be read as clearly, easily and fast as possible, but the truth is, sometimes they can get extensive.

In some IDEs, this can cause the comment to overflow, looking unprofessional:

Code

```
# Declare a list variable that will serve to store dictionaries. This is crucial since we won't
be able to retrieve the objects otherwise.
mylist = []
```

Breaking the comment in two or more lines improves readability:

Code

```
# Declare a list variable that will serve to store dictionaries.
# This is crucial since we won't be able to retrieve the objects otherwise.
mylist = []
```

We can also use a docstring if the comment is too long. We just have to make sure to keep format consistency across our annotations:

Code

```
'''
Declare a list variable that will serve to store dictionaries.
This is crucial since we won't be able to retrieve the objects otherwise.
'''
mylist = []
```

2.4 Section definition

When working with long scripts, we can also use the hash symbol `#` to define section separators and divide our code blocks by including some delimiter along with the comment.

The format used is purely based on personal preference but should generally be a collection of uniform characters typically used to delimit:

Code

```
# ------------------------------
# ------------------------------
# This denotes a section start point
# ------------------------------
# ------------------------------

# ------------------------------
# ------------------------------
# This denotes a section end point
# ------------------------------
# ------------------------------

# ------------------------------
# This denotes a subsection start point
# ------------------------------

# ------------------------------
# This denotes a subsection end point
# ------------------------------

# This denotes a subtitle
# ------------------------------
```

We can also use a different, more emphatic character combination to denote a centered title:

Code

```
# +----------------------------------------+
# |      This denotes a centered title     |
# +----------------------------------------+
```

The important thing to remember is not to overcrowd our code. Otherwise, it could become illegible.

Also, setting a section spacing standard can help.

3. Docstrings

We already used docstrings to insert simple comments, but the main reason they were created was to be used inside functions to explain what the object does and its expected inputs & outputs.

This is especially relevant when writing extensive code. Also, it provides a way for others, or even ourselves, to use the function as a modular object and know exactly what to expect from a function call.

We can include a docstring inside a function by using the same format as before:

Code

```
def addAges(age1, age2, age3):
    '''
    Parameters
    ----------
    age1 : int
        Age 1, from 1 to 10.
    age2 : int
        Age 2, from 11 to 20.
    age3 : int
        Age 3, from 21 to 30.

    Returns
    -------
    sumOfAges : int
        Sum of ages 1, 2 and 3.
    '''

    sumOfAges = age1 + age2 + age3

    return sumOfAges
```

One crucial detail to remember, is that docstrings are indentation-sensitive, meaning an improperly indented docstring will throw an `IndentationError` upon execution:

Code

```
def addAges(age1, age2, age3):
'''
Parameters
----------
age1 : int
    Age 1, from 1 to 10.
age2 : int
    Age 2, from 11 to 20.
age3 : int
    Age 3, from 21 to 30.

Returns
-------
sumOfAges : int
    Sum of ages 1, 2 and 3.
'''

    sumOfAges = age1 + age2 + age3

    return sumOfAges
```

Output

```
IndentationError: expected an indented block after function definition on line 1
```

4. Indentation

Indentation is used in all programming languages and has two primary purposes depending on the language used:

- As a way to improve code readability
- As part of the actual syntaxis

Python uses indentation as part of its syntaxis. This is why the `IndentationError` above was raised.

Even though Python code is generally not executed without the proper indentation, we still can indent incorrectly in some cases without an error being raised. This is common when using argument continuation in new lines:

Code

```
mylist1 = [1, 2, 3, 4,
     5, 6, 7, 8]

mylist2 = [1, 2, 3, 4,
            5, 6, 7, 8]
```

The above code runs fine but doesn't look fine. In fact, it seems as if something were off. The reason is that, even though Python does not raise an `IndentationError`, the indentation is incorrect in both cases.

We can properly indent our lists by aligning parenthesis, or in this case, brackets:

Code

```
mylist1 = [1, 2, 3, 4,
           5, 6, 7, 8]

mylist2 = [1, 2, 3, 4,
           5, 6, 7, 8]
```

5. Line breaks, parenthesis, brackets & curly brackets

5.1 Line breaks

Line breaks are useful but rare, often reducing the code's legibility. They can also lead to syntax errors if used sparingly.

We can use them for specific cases, *e.g. whenever we are presented with a variable containing multiple characters*.

A line break can be achieved by adding a backslash `\` to the section we want to break:

Code

```
myvar = 1 + 2 + 3 + 4 + 5 + 6 + 7 + 8 + \
        9 + 10 + 11 + 12
```

5.2 Parenthesis

A better alternative to backslashes would be to enclose our variable arguments in parenthesis. This also allows us to continue writing on a new line:

Code

```
myvar = (1 + 2 + 3 + 4 + 5 + 6 + 7 + 8 +
         9 + 10 + 11 + 12)
```

5.3 Brackets & curly brackets

As we saw before, we can enclose arguments in parenthesis to continue writing on a new line.

We can also continue arguments on a new line if we're dealing with other objects such as lists, arrays or dictionaries.

This is extremely useful when for example, we have to specify a high-dimension array with multiple nested brackets.

If we were to attempt a one-liner, the code would become challenging to read:

Code

```
import numpy as np

# Declare a 3-dimensional numpy array
myarr = np.array([[[10,20,30,40], [50,60,70,80]]])
```

We could use line continuation to break each specific dimension into a new line:

Code

```
import numpy as np

myarr = np.array([
                    [
                        [10,20,30,40],
                        [50,60,70,80]
                    ]
                ])
```

Another example would be a JSON-like formatted object:

Code

```
schema = {'type': 'record','name': 'dataset','namespace': 'dataset','doc': 'This schema consists
of 1 int type and 7 string types','fields': [{'name': 'Name', 'type': 'string'},{'name': 'Age',
'type': 'int'},{'name': 'Occupation', 'type': 'string'},{'name': 'Country', 'type': 'string'},
{'name': 'State', 'type': 'string'},{'name': 'City', 'type': 'string'}]}
```

This is frankly unpleasant and overflows the document width by about three times.

We can do better by using line continuation:

Code

```
schema = {
    'type': 'record',
    'name': 'dataset',
    'namespace': 'dataset',
    'doc': 'This schema consists of 1 int type and 7 string types',
    'fields': [
        {'name': 'Name', 'type': 'string'},
        {'name': 'Age', 'type': 'int'},
        {'name': 'Occupation', 'type': 'string'},
        {'name': 'Country', 'type': 'string'},
        {'name': 'State', 'type': 'string'},
        {'name': 'City', 'type': 'string'}
    ]
}
```

6. Appropriate variable naming

When we start writing lengthier programs, it's inevitable to start losing count of the variables we declare. Or maybe we have excellent memorization skills and don't really care what names we use, but then Juan from Engineering, the poor guy responsible for deploying our code, sends us an email begging to send a variable definition document because he cannot understand a thing:

Code

```
styaway = (3, 1, 4.3, 6.5, 2)
Gobbledegook = [4, 3, 5, 6, 7, 3]
theCollywobbles = [1, 2, 3, 4, 5]
theothercollywobbles = [4, 3, 6, 7, 8, 2]
mthBreather = [8, 7, 1, 4, 5, 6]
KNickers = (1, 2, 3)
xyz = 3.1416

for w in range(len(mthBreather)):
    theothercollywobbles.append(w)
    for d in theCollywobbles:
        print(xyz)
    copyofthecollywobbles = theCollywobbles.copy()
```

Utterly dreadful, right?

We can do much better and give Juan a break:

Code

```
myTuple = (3, 1, 4.3, 6.5, 2)
myTuple_2 = (1, 2, 3)
myList = [4, 3, 5, 6, 7, 3]
myList_2 = [1, 2, 3, 4, 5]
myList_3 = [4, 3, 6, 7, 8, 2]
myList_4 = [8, 7, 1, 4, 5, 6]
pi = 3.1416

for constants in range(len(myList_4)):
    myList_3.append(constants)

    for items in myList_2:
        print(pi)

    myList_2_copy = myList_2.copy()
```

7. Spacing

Lastly, spacing consistently and adequately will always improve our code. Spacing creates a sense of separation and independence and instantly provides a better reading experience.

Python does not require us to insert spaces between characters, but if we leave our code without spacing, or even worse, with inconsistent spacing, it will be much harder to read:

Code

```
myList = [1,2,3,4,5,6,  7, 8,9,10, 11]
```

The line above generates our list object successfully and without spaces in between, of course, but the reading is a challenge on its own.

This is why it's always good practice to be consistent about the spaces we use, how we use them, and why we use them.

We can implement single spaces between our list arguments:

Code

```
myList = [1, 2, 3, 4, 5, 6, 7, 8, 9, 10, 11]
```

We can also implement new lines to make distinctions between different segments or operations:

Code

```
def myFun(x, y, z):
    x =+ 1
    y *= 1
    z -+ 1
    t = 3

    for i in range(x):
        t -= x

    return t
```

§

Modularization & scalability

1. Modularization

Code is meant to be modular. Sure, we can write a single script to bulk-rename *n* number of files. That script would only be used for that purpose and would not require any dependence or be required by any other program.

But the reality is that most of the code that is written is meant to be used in conjunction with other code; it's intended to work as part of a bigger system. This is called modularization and is an extremely powerful concept in programming.

In Python, modularization can be achieved by using different levels of abstraction, such as functions, classes and other scripts.

Let's take an example where we want to calculate the sum of all numbers inside a `list` object:

Code

```
nums = [1, 2, 3, 4, 5]

total = 0

for x in nums:
    total += x

print(f'Sum of values is: {total}')
```

Output

```
Sum of values is: 15
```

This code performs as expected:

- It first defines our list of integer numbers.
- It then defines our initial counter `total` and sets it to 0.

- It then iterates over all elements of our `nums` list, adding each element to our counter `total`.
- Finally, it prints some string along with the sum of all numbers.

Still, it presents the following disadvantages:

- The code is only usable inside the script itself, meaning we cannot call it from another script.
- We must redefine our `nums` object each time we want to perform a different calculation.
- We cannot assign the `total` calculation to another variable unless we explicitly copy the `total` variable.
- There's no way for the user to input a custom list of numbers if the script is run from a shell.

This code can be modularized by refactoring it to a function:

Code

```
def getSum(nums):
    total = 0

    for x in nums:
        total += x

    print(f'Sum of values is: {total}')

    return total
```

We can then call our function and assign the returned variable `total` to *n* number of variables:

Code

```
nums_1 = [1, 2, 3, 4, 5]
nums_2 = [6, 7, 8, 9, 10]
nums_3 = [11, 12, 13, 14, 15]
nums_4 = [16, 17, 18, 19, 20]

total_1 = getSum(nums_1)
total_2 = getSum(nums_2)
total_3 = getSum(nums_3)
total_4 = getSum(nums_4)

print(total_1, total_2, total_3, total_4)
```

Output

```
Sum of values is: 15
Sum of values is: 40
Sum of values is: 65
Sum of values is: 90
15 40 65 90
```

Finally, we can call our function from an entirely different script by importing `getSums()` as a custom method. For this, we need to create a new Python script, `my_fun.py`, in the same directory as our current script and include our `getSum` function:

Code

```
from my_fun import getSum

nums_5 = [-3, -2, -1, 0, 1, 2, 3]
total_5 = getSum(nums_5)
```

Output

```
Sum of values is: 0
```

We can even set an alias for our `getSum` function upon importing:

Code

```
from my_fun import getSum as sumVals

nums_6 = [-3, -2, -1, 0, 1, 2, 3]
total_5 = sumVals(nums_6)
```

Output

```
Sum of values is: 0
```

This is just scratching the surface of what abstraction can do. A more detailed study of other objects is out of the scope of this article, but not to worry, we will get there eventually.

2. Scalability

A scalable code does not require frequent modifications to maintain performance when handling varying workloads. This can be achieved from several viewpoints and strongly depends on what the code is intended to perform.

When handling large data sets, for example, it is important to think of the following:

- Properly define the file format(s) to be used.
- Properly define a schema to be used and keep consistency.
- Properly define the objects that will be used to store the information.
- Properly define the aggregation levels that will be taking place.
- Manage memory according to the expected data set size.
    - A simple `del` after using an object will clear it from memory.

Further information on Big Data file formats can be found in my 3-article series 6 Big Data File Formats Compared.

This is just one example, but there are multiple measures that can be taken to secure scalability and execution integrity.

§

Performance

Although this area requires a little bit of experience and additional knowledge in algorithmic design & computational complexity, we can at least perform the basics to ensure our code does what it's supposed to in the least amount of time possible, consuming the least amount of resources. This is called **refactoring** and is crucial, especially when productionizing code.

We want to be efficient with the execution times & memory management while performing the same operation.

Of course, there are countless variables we can optimize, such as using a faster programming language for starters. Nonetheless, we can focus on the most basic mechanisms we can implement while still using Python as our trusty fellow.

1. Proper module import

If we are using a single module from a vast library, why import the whole thing? We can refer to this principle as the *bring just what you need* approach.

When we import a module, the Python interpreter has to look for it and then load it into the cache. Suppose we import the whole library without specifying the module we're using. In that case, all modules belonging to that library are saved on the cache and are ready to be employed upon a user request:

Code

```
import numpy as np

myarr = np.array([1, 2, 3])
```

This is unnecessary and takes cache space that should not be taken in the first place.

We can instead import a specific module by explicitly declaring it upon importing the library:

Code

```
from numpy import array as arr
```

This method is also helpful since we won't call the `array` method using the `np.array` form. Instead, we can call it directly as `arr`:

Code

```
from numpy import array as arr

myarr = arr([1, 2, 3])
```

Of course, there are some caveats to this, the main one being we can get confused about where the `arr` method came from. We can reduce this by using meaningful names as our method aliases.

2. Built-in libraries over external ones

As stated before, when we import a module, the Python interpreter performs, in a general way, a search consisting of the following steps:

1. Search in the `sys.modules` dictionary
    1. If found, return it.
    2. If not found, call `find_spec()` on each `sys.meta_path`
        1. If spec is found, create it and add it to `sys.modules`

By default, built-in Python methods are already located in `sys.modules`. This means that the Python interpreter will find it during the first step.

In contrast, if we import an external module for the first time, Python will have to perform the remaining steps, which takes additional time.

This is not a big deal since it only happens the first time we import a module. Nonetheless, it's always a better option to use built-in modules in favour of external ones if the built-in option has better performance.

Also, built-in modules will probably be more common among fellow programmers, thus skipping the step of installing a new library just for our code.

Finally, built-in modules are written in `C`, whereas external modules can be written in slower languages such as Python.

3. Proper data structures

Some Python data structures are used more often than others due to their popularity and ease of manipulation. For example, tuples, lists and dictionaries are all prevalent data structures and can be found on virtually any Python code.

The thing about clinging to a fixed data structure for everything is that sometimes that option may underperform in specific tasks when compared to other alternatives:

- Instead of using a `list`, we can employ a `set`, which is implemented using *hash tables*, and offers better performance on tasks involving searching.
- If we are not mutating the values inside our `list`, we can even use a `tuple` as a faster alternative.
- If we have a collection of homogeneous data types, we can also use an `array` instead of a `list` for better performance in some applications.
- Instead of using the traditional `dict` data structure, we can use the external `microdict` alternative based on hash tables.
- Instead of using a `pandas.DataFrame`, we can use an `array` if we plan on performing linear algebra operations. This would also provide better performance.

4. List comprehensions over `for` loops

List comprehensions are syntactic constructs that allow us to create lists based on the values of another list. `for` loops can do the same but present higher execution times.

One thing to bear in mind is that although list comprehensions are valuable, they can and will reduce readability for people not accustomed to their single-line structure. This is why it's bad practice to include more than two or three nested arguments in list comprehensions (*we must also remember that Python code is meant to be highly readable, and we don't want to take that away*):

Code

```
# Conventional for loop
mylist_1 = []
for i in range(20):
    mylist_1.append(i)

# List comprehension
mylist_2 = [i for i in range(20)]

# Print resulting objects
print(mylist_1)
print(mylist_2)
```

Output

```
[0, 1, 2, 3, 4]
[0, 1, 2, 3, 4]
```

We can actually compare execution times by using a larger `range`:

Code

```
import time

num_trials = 100000000

# Conventional for loop
start = time.time()
mylist_1 = []

for i in range(num_trials):
    mylist_1.append(i)

end = time.time()
print('For loop execution time:', end-start, 'seconds')

# List comprehension
start = time.time()

mylist_2 = [i for i in range(num_trials)]

end = time.time()

print('List comprehension execution time:', end-start, 'seconds')
```

Output

```
For loop execution time: 4.528107166290283 seconds
List comprehension execution time: 2.702324628829956 seconds
```

5. `for` loops vs. `while` loops

Each loop method has its advantages and disadvantages. It's just a matter of evaluating which one suits our needs best.

`while` loops, while incredibly powerful, can sometimes underperform their `for` loop counterpart (*and we say sometimes because there are times when a* `while` *loop is a better alternative*).

On the other hand, `for` loops, as opposed to `while` loops, don't have to verify a specified condition to continue, making their execution faster.

They can also iterate through a series of values using the `range()` function, implemented in `C`, and thus much faster.

6. Multiple variable assignment

When we're assigning multiple variables, we have two options:

- Assign each of them using a separate new line.
- Assign them all at once in a single line.

Of course, if we have multiple variables to assign, the first approach would offer significantly better readability than the second one. Still, if we're dealing with a small set of variables, we can use the one-line approach:

Code

```
# New line variable assignment
myvar_1 = 1
myvar_2 = 2
myvar_3 = 3
myvar_4 = 4
myvar_5 = 5

# Single line variable assignment
myvar_1, myvar_2, myvar_3, myvar_4, myvar_5 = 1, 2, 3, 4, 5
```

Both methods produce the same result. The one-line approach simply saves us some space and can often improve readability.

7. Using decorators

This is a more advanced topic and requires us to learn what a decorator is in the first place. A decorator is a function that takes another function as an argument, extends the behaviour of the latter without explicitly modifying it, and returns a new, decorated function.

This sounds confusing, but we can simplify it by using an example:

Let us imagine we have a function `myFun`, which accepts two integer numbers and prints them:

Code

```
# Define a simple function
def myFun(x, y):
    print(x, y)

myFun(7, 77)
```

Output

```
7 77
```

Let us now imagine that we would like to extend the behaviour of `myFun`, because our function is too simple, but we don't want to change its syntax in any way.

We can define a decorator function `decoratorFun`:

Code

```
# Define a decorator function
def decoratorFun(decoratedFun):

    def wrapperFun(*args, **kwargs):
        print('Start decorator')
        decoratedFun(*args, **kwargs)
        print('End decorator')

    return wrapperFun
```

Our `decoratorFun` prints two strings:

- `Start decorator`
- `End decorator`

What we want to do now is wrap our original function `myFun` print statement around our decorator function `decoratorFun` strings. We can add a `@decoratorFun` indicator on top:

Code

```
@decoratorFun
# Define a simple function
def myFun(x, y):
    print(x, y)

myFun(7, 77)
```

Output

```
Start decorator
7 77
End decorator
```

We have effectively extended the functionality of our simple function `myFun` without changing anything (*not even its arguments*).

Of course, this is as simple as it gets, but we can imagine using decorators involving more complex functions and classes.

§

Testing & debugging

1. Exception handling

If we go back to the Docstrings section and take a closer look at the function we defined, there is nothing in the code impeding the user from inputting a wrong parameter. Sure, we defined a docstring telling the user what to do, but we did nothing to ensure how the function would behave if the user gave an integer number out of the specification bounds as input or, even worse, the wrong type.

In the best-case scenario, the user reads the docstring and understands what to do. In the worst-case scenario, the user doesn't even know what a data type is and inputs something like this:

Code

```
def addAges(age1, age2, age3):
    '''
    Parameters
    ----------
    age1 : int
        Age 1, from 1 to 10.
    age2 : int
        Age 2, from 11 to 20.
    age3 : int
        Age 3, from 21 to 30.

    Returns
    -------
    sumOfAges : int
        Sum of ages 1, 2 and 3.
    '''

    sumOfAges = age1 + age2 + age3

    return sumOfAges

addAges('1', '1', '0')
```

What would happen? Well, the first thing to bear in mind is that if we input garbage, we get garbage. That is if we don't have a handler for these types of slips (*which happens often and can easily break a precarious program*).

Fortunately, with this particular input, the program would not break. Instead, it would simply concatenate the three ages (*not sure what's worse*):

Output

```
'110'
```

We can see that not only is the intended operation wrong, but the returned object is a string.

The user could slip more subtly and input something like this:

Code

```
addAges('1', 2, 3)
```

Output

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

And there we have it, lads; the user broke our program in 2 tries. He even got upset because "*some unintelligible nonsense appeared on his screen, even though he did everything right*". Good luck explaining that to our boss.

We could've avoided this by using exception handlers.

These handy methods allow us to redirect errors such as the one we encountered and do something productive with them. A custom exception can:

- Return a more explanative message error to the user or even the programmer.
- Prevent the program from breaking by bypassing the error and redirecting the flow.
- Try to catch the error beforehand, and actually try to fix it.

The catch is, formulating appropriate exceptions is more challenging than it looks because we have to think outside the box; we have to consider every possible way our program could break. This methodology is called *Test to Fail*.

Exception handling is so critical that a poorly-written program with incorrect exception handling could quickly become vulnerable to attacks (*an SQL injection can be prevented by proper input exception handling*).

The logical way an exception works, in its most simple way, is as follows:

- We know what the user might input.
- If the input is wrong in *this* way, perform *that* action.
- If the input is wrong in *this* other way, perform *that* other action.

We can translate a simple example into code:

Code

```
def inputNum(num):
    '''
    Parameters
    ----------
    num : int
        Num, a real integer number.

    Returns
    -------
    num_s : str
        num as string type.
    '''
    if type(num) != int:
        raise TypeError('Please input a real integer number')

    else:
        num_s = str(num)

    return num_s

inputNum('a')
```

Here we have implemented an exception handler to ensure that the user is inputting the correct data type, in this case, a real integer number.

Output

```
TypeError: Please input a real integer number
```

We can level up a notch and actually try to correct the input by using a `try` `except` combination:

Code

```
def inputNum(num):
    '''
    Parameters
    ----------
    num : int
        Num, a real integer number.

    Returns
    -------
    num_s : str
        num as string type.
    '''
    if type(num) != int:
        try:
            num_i = int(num)
            num_s = str(num_i)
        except ValueError:
            raise TypeError('Please input a real integer number')

    else:
        num_s = str(num)

    return num_s

inputNum('a')
```

Output

```
TypeError: Please input a real integer number
```

Of course, we will never be able to cast a string type to an integer type by using this simple piece of code. However, we can cast a float type to an integer type:

Code

```
inputNum(3.56)
```

Output

```
'3'
```

With this simple step, we've not only impeded a significant catastrophe but also made our code more flexible, letting the user input float types and get the same answer. We just need to be careful to include what the `int()` method will actually do. In our case, it truncates the float number and does not round it up to the nearest integer.

Going back to our original example, we can include exception handlers to ensure that the user knows what to input:

Code

```
def addAges(age1, age2, age3):
    '''
    Parameters
    ----------
    age1 : int
        Age 1, from 1 to 10.
    age2 : int
        Age 2, from 11 to 20.
    age3 : int
        Age 3, from 21 to 30.

    Returns
    -------
    sumOfAges : int
        Sum of ages 1, 2 and 3.
    '''
    if (type(age1) != int) | (type(age2) != int) | (type(age3) != int):
        raise TypeError('Please input an integer number.')

    elif (age1 <= 0) | (age2 <= 0) | (age3 <= 0):
        raise AttributeError('All numbers have to be positive and non-zero.')

    elif (age1 < 10) | (age2 < 20) | (age3 < 30):
        raise AttributeError('Age 1 must be between 1 and 10. '\
                             'Age 2 must be between 11 and 20. '\
                             'Age 3 must be between 21 and 30. ')

    else:
        sumOfAges = age1 + age2 + age3

    return sumOfAges
```

These are just some examples; we could improve it even further, but the point is made.

The one thing we must remember about this section is that good code will not only have a significant portion dedicated to handling errors but will do it appropriately and elegantly.

2. Print statements

Print statements allow us to make fewer typing and logical mistakes by outputting useful messages about specific sections. We can use print statements to debug or output messages to the anxious user.

Even though we sometimes feel like printing every step to ensure we're not losing it, we should moderate ourselves and not go too bananas; otherwise, we might turn an anxious user into an angry user.

Print statements are intended to inform about important processes in our code, *e.g. for each iteration of a loop, print the file number we're reading, along with the total number of files pending to read.*

They can also be used to debug specific sections, specifically when working with functions and have no visibility of the values assigned inside.

The key to remember is to avoid abusing them and remove any debugging print statements we used when signing off our code to another person.

§

Dependency handling

Lastly, dependencies play a huge role in avoiding execution errors and ensuring maintainability. There are multiple mechanisms we can put into place to ensure that our code will run smoothly across environments:

- Specify the dependencies required in a `readme` file, and pack it along with our code. Also, add it to the project repository if we're using version control.
- Create a `.yml` file to specify and handle the required environment.
- Make sure the packages we're using are up-to-date and actively maintained.
    - This is not always possible, but we can try to include the least amount of unmaintained packages in our code to ensure the least amount of conflicts.
- If we create a proprietary package we're using for some project, we need to keep the file handy and, if possible, upload it to a version control software such as GitHub.
- Make sure we're using the correct methods and that they'll not be deprecated soon.
    - This is an easy one. Most methods output a warning if they're about to be deprecated in future versions. We need to be proactive with these warnings and substitute the old methods in favour of updated ones before it's too late and our angry user turns into an exasperated user.

§

Conclusions

We've reviewed multiple yet simple mechanisms we can employ to make our code cleaner, more elegant, modular, usable, scalable and safer. These measures can not only help us become better programmers but better collaborators. It will make reading code a pleasure instead of an agonizing process and instantly boost our credibility.

§

References

- Python Documentation, Built-in Exceptions
- Python Documentation, Errors & Exceptions
- Towards Data Science, What happens when you import a Python module?
- Towards Data Science, 3 data structures for faster Python Lists

§

Copyright