Why Python versus Bash (reasons for SRE)

Bash Shell Scripting is a very powerful language, especially thanks to the pipes, that allow processes to communicate and redirect their flow from STDIN, STDOUT, STDERR.
But:

- Bash has not been thought to be a modern structured language, and does not support OOP
- Does not support Unit testing
- Bash is very slow compared to Python.
- Complex programs simply cannot be written using Bash, as it relays on the execution of external programs and data parsing it doesn't handle well unexpected errors
- Doesn't support Exceptions, only error levels.
- Another huge problem from Bash is the lack of support for Decimals.
- Functions do not return values, so you have to use and update global variables (and that's ugly and risky)
- Bash scripts are for Linux/Unix, they are not portable to Windows

In the other hand Python:

- has many built-in libraries for all kind of tasks
- is portable across different Operating Systems (Linux, Mac Os X, Windows).
- Supports OOP, Unit Testing
- Proper Debugging, breakpoints, inspects… and is integrated in IDEs like PyCharm
- Can be easily connected with Databases, MQS[15]...

OOP vs Procedural code (SRE)

When a person that comes from System Administration background, using Bash scripts, starts to program in Python, it is easy to use the same routines. Like:

- Using vi or emacs in text mode instead of a modern IDE
- Programming Procedural, not using OOP[16]
- Not doing Unit Testing

Those are things that have to be improved.
Modern IDEs have Syntax Checking, Dependency Checking, Code Coverage, Debugger integrated, Git Support, safe Refactors, unused variables detection…

There are many ways to achieve the same result, and one can work using Procedural. Object Oriented Programming is another pattern, or another paradigm, a tool, if you want to achieve our results. But we have to learn those patterns in order to be able to take advantage of them and use the best solution for our problems.
Believe, using Procedural there are things that you will need a lot of "if", arrays, etc… until getting to a point where it is unmaintenable; just to try to achieve the same that you'll achieve with OOP and inheritance easily.
So I consider very important that you get to understand why we use OOP.

15 Message Queue Systems like RabbitMQ
16 OOP means Object Oriented Programming

The best explanation I did read, many years ago, is that programs use Data, and Code to manipulate the data and a class is like a ball, a unit, that globes the Data and the Code to manipulate the Data inside the class.
The class is the blueprint, and an instance is when we produce an object based on that blueprint.

For example, for cataloging all our Servers we could have a class Server with the code that will return the Model, P/N, amount of RAM, year we bought it... and with code to update the amount of RAM, update the Storage.

The Inheritance[17] allows us to write a base class, and have classes based on, that extend their functionality.
An example would be the code for an E-Commerce System present in 10 countries. Instead of having 10 different projects, or a main project full of **if**'s to act different depending on the country. For example Germany has certain payment methods that other countries don't, and Brazil has "parcelados" for allowing to split a Visa payment across several months.
So for example we would create a class called EcommerceSystemGermany extends BaseEcommerceSystem, and only the code that is different respect the base Ecommerce System, would be written. For example, a method like show_payment_methods(), method pay(), etc.. Those methods will overwrite the methods of BaseEcommerceSystem, and we can also add new methods, extending the functionality.

17 https://en.wikipedia.org/wiki/Inheritance_(object-oriented_programming)

A practical sample of OOP for the SRE world

Imagine we want to write a program allowing us to manage the Commodity Servers of the Company that are in the Data Center.

We define a class Server like this:

server.py
https://gitlab.com/carles.mateo/python_combat_guide/-/blob/master/src/server.py

```
import os
import subprocess
import time


class Server:
   s_hostname = ""
   s_user = ""
   s_kernel_version = ""
   s_administration_ip = ""
   s_status = ""

   def __init__(self, s_hostname, s_administration_ip, s_user):
       self.s_hostname = s_hostname
       self.s_user = s_user
       self.s_administration_ip = s_administration_ip
       self.s_kernel_version = self.update_kernel_info()

   def send_poweroff(self):
       self.s_status = "POWERING OFF"
       s_command = "ssh " + self.s_user + "@" + self.s_administration_ip + "
'poweroff'"
       os.system(s_command)
       time.sleep(10)
       self.s_status = "POWERED OFF"

   def send_reboot(self):
       self.s_status = "REBOOTING"
       s_command = "ssh " + self.s_user + "@" + self.s_administration_ip + "
'reboot'"
       os.system(s_command)
       time.sleep(10)
       self.s_status = "POWERED ON"

   def update_kernel_info(self):
       self.s_status = "GETTING SYSTEM INFO"
       # Enforcing the use of a specific Key for Server
       s_cert_path = "/var/certs/id_rsa-" + self.s_administration_ip
       o_ssh = subprocess.Popen(["ssh", "-i " + s_cert_path, self.s_user + "@" +
self.s_administration_ip],
                                stdin=subprocess.PIPE,
                                stdout=subprocess.PIPE,
                                stderr=subprocess.PIPE,
                                universal_newlines=True,
                                bufsize=0)

       # Send ssh commands to stdin
       o_ssh.stdin.write("uname --kernel-release\n")
       o_ssh.stdin.close()
```

Coding

General Style

PEP-8

We base in the PEP-8 Standard.
https://www.python.org/dev/peps/pep-0008/

Some philosophy:
https://www.python.org/dev/peps/pep-0020/

Although we use some differences as PEP-8 dates from 05-Jul-2001, with some parts updated up to 01-Aug-2013, and certain statements are simply obsolete.
For example, we don't enforce 70 or 100 columns width as Maximum Line Length, cause we use modern tools such as Full HD monitors and modern IDEs for development and so 100 columns is too low, and not practical.
Even 120 is too low given the wide good screens we use, however we need to enforce a limit in order to make automated programs that parse the code work correctly.
So we enforce a limit of 120 characters from line. Any line longer, whatever is commands, calls to methods, etc… should be split in several lines following the PEP-8 Standard.

MT Notation

We adopted other improvements.
We use the MT Notation variables prefix:
https://blog.carlesmateo.com/mt-notation-for-python/

Quick tips about MT:

Prefix	Description of the variable
s_	String
i_	Integer
f_	Float
b_	Boolean
o_	Object / Class / Resource

Some other prefixes may be added to this:

Prefix	Description of the variable
h_	Array Hash, also called dictionary
a_	Array List
t_	Tuples. Like Lists, but immutables
c_	Counters. From `Collections.Counter`
by_	Byte object
ba_	ByteArray object

In some cases a prefix for a local variable, when we want to differentiate from local scope, will be added.

We can be even more precise if we work with exact types, so for example, if we have a dictionary for employees with Integer keys and String values we could use:
d_i_s_employees

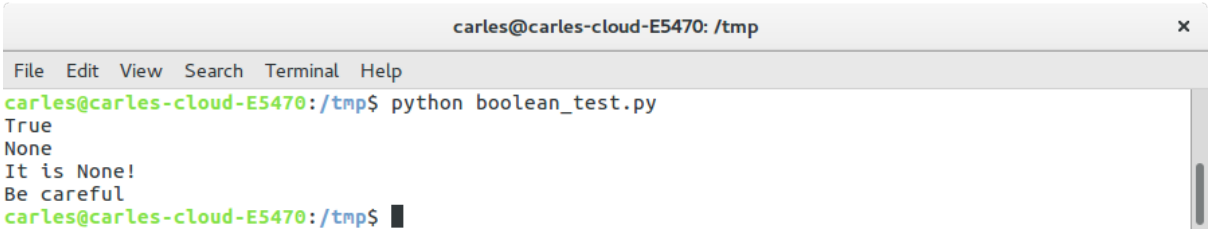
Prefix	Description of the variable
l_ (optional)	Local variable. In order to avoid shadowing global variables. **Note:** is an L in lowercase. For instance: **l_o_subprocess** inside a method or function in opposition to the **p_o_subprocess** or **o_subprocess** from the global scope. I don't recommend to use **l_** as overcomplicates things. And you should not use global variables.
p_ (recommended)	If you use public variables, so from a global scope, like visible for all the package, it is recommended that you define them with prefix **p_**. For example: p_username
CONSTANTS	Global Constants that are immutable, can be defined in uppercase. For example: PATH_ETC = '/etc' Or: s_PATH_ETC = '/etc'

Return

Methods will be consistent returning a type.
Preferentially they will return at least a boolean for indicating if the operation was successful.
If the method returns a boolean **True**, we will be consistent and return **False** in the other cases.
We will not just return with **return**, cause this causes **`return None`** for real.

Take a look at this demonstration:

```
# Proof of Concept for avoiding return without the type
# Author: Carles Mateo
# Creation Date: 2018-03-27
#

from pprint import pprint


def boolean_test(b_value):
   if b_value is False:
       return

   return True


b_true = boolean_test(True)
b_false = boolean_test(False)

pprint(b_true)
pprint(b_false)

if b_false is False:
   print "I detect it as False (even if it's None)"

if b_false is True:
   print "I detect it as True (even if it's None)"

if b_false is None:
   print "It is None!"


print "Be careful"
```



Error code in return

Python allows the return of several parameters, so we take advantage of this.
Any call to a method that returns data should return an error code.
0 for everything working Ok, other numeric values for errors.

```
        i_error_code = int(s_error_code)
        s_output = "".join(s_output.split("\n")[1:])
    except:
        i_error_code = 99
        s_output = ""

    return i_error_code, s_output
```

In many cases it is worth it and the preferred choice to return:
b_success (Boolean), i_error code (Integer), s_output (String)

Boolean to indicated the success or failure of the operation.
Integer to indicate the error code of the operation.
String if further error message, or output info is expected.

So the calling method only has to check if the operation succeeded or failed, by checking the Boolean returned.

Predictability in return: Return always the same

Return should be predictable, and always return the same types.
I have seen a method that in some cases would return an string, and in other a list of strings.
That's wrong, as will generate errors in the calling side.
In this case the method should return always a list, even if it has a single string, or an empty string, of even if returns an empty list.

Files

Always in lowercase.
I.e.:

So take a look at this code[41] which uses a list:

```
def add_instance(s_instance_uuid, a_instances=[]):
    a_instances.append(s_instance_uuid)
    return a_instances

print(add_instance("267f2d5a-8930-11ea-bea8-2b67fbe5ab05"))
print(add_instance("2733dac0-8930-11ea-906f-9b6e24c72f0b"))
print(add_instance("278c9b56-8930-11ea-9539-efd8d07f289a"))
```

This outputs:

```
['267f2d5a-8930-11ea-bea8-2b67fbe5ab05']
['267f2d5a-8930-11ea-bea8-2b67fbe5ab05', '2733dac0-8930-11ea-906f-
9b6e24c72f0b']
['267f2d5a-8930-11ea-bea8-2b67fbe5ab05', '2733dac0-8930-11ea-906f-
9b6e24c72f0b', '278c9b56-8930-11ea-9539-efd8d07f289a']
```

As the documentation page specifies[42], here is a trick to prevent this behavior of sharing mutables between subsequent calls:

```
def f(a, L=None):
    if L is None:
        L = []
    L.append(a)
    return L
```

Keyword Argument for a Method/Function

I'm not very fan of this cause is confusing for the Juniors and errors may be introduced but is good that you know that is possible.

When the ending parameter has two asterisks `**, in front of the name`, it receives a dictionary with the keyword and value, except for those that correspond to a formal parameter already defined in the signature.

This may be combined with a single asterisk * preceding a formal parameter `name` which receives a tuple containing the positional arguments beyond the formal parameter list. (`*name` must occur before `**name`.)

So to understand it, nothing better than sample code:

41 https://gitlab.com/carles.mateo/python_combat_guide/-/blob/master/src/wow_instances.py
42 https://docs.python.org/dev/tutorial/controlflow.html#default-argument-values

https://gitlab.com/carles.mateo/python_combat_guide/-/blob/master/src/keyword.py

```
#
# keyword.py
#
# Author: Carles Mateo
# Creation Date: 2020-04-28 18:50 GMT+1
# Description: How the keyword arguments work
#
class GameAdder:
    h_games = {}
    def add_game(self, s_game_name="Unknown", args=None, **kwargs):
        print("Preparing parameters for: " + s_game_name)
        print("-"*40)
        print(kwargs)
        print("-" * 40)
        print("Filtering out Private Variables")
        print("-" * 40)
        for s_key in kwargs:
            if s_key[0] != "_":
                if s_game_name in self.h_games:
                    self.h_games[s_game_name][s_key] = kwargs[s_key]
                else:
                    self.h_games[s_game_name] = {}
                    self.h_games[s_game_name][s_key] = kwargs[s_key]
            else:
                print("Private (internal) property: " + s_key)
    def get_games(self):
        return self.h_games
o_ga = GameAdder()
o_ga.add_game(s_game_name="Wow",
              _release_ver = "7.1050",
              _release_date = "2020-04-29",
              s_nice_name = "World of Warcraft")
o_ga.add_game(s_game_name="OW",
              _release_ver = "7.7777",
              _release_date = "2018-01-10",
              _responsible = "Carles Mateo",
              i_price = 35)
o_ga.add_game(s_game_name="SC2",
              _release_ver = "5.1234",
              _release_date = "2010-02-01",
              s_responsible = "Carles Mateo",
              i_price = 50,
              b_is_free=True)
print("\n")
print("Games")
print("=====")
print(o_ga.h_games)
```

The output is:

```
carles@fast:~/Desktop/code/python_combat_guide/src$ python3 keyword.py
Preparing parameters for: Wow
----------------------------------------
{'_release_ver': '7.1050', '_release_date': '2020-04-29', 's_nice_name':
'World of Warcraft'}
----------------------------------------
Filtering out Private Variables
----------------------------------------
Private (internal) property: _release_ver
Private (internal) property: _release_date
Preparing parameters for: OW
----------------------------------------
{'_release_ver': '7.7777', '_release_date': '2018-01-10', '_responsible':
'Carles Mateo', 'i_price': 35}
----------------------------------------
Filtering out Private Variables
----------------------------------------
Private (internal) property: _release_ver
Private (internal) property: _release_date
Private (internal) property: _responsible
Preparing parameters for: SC2
----------------------------------------
{'_release_ver': '5.1234', '_release_date': '2010-02-01', 's_responsible':
'Carles Mateo', 'i_price': 50, 'b_is_free': True}
----------------------------------------
Filtering out Private Variables
----------------------------------------
Private (internal) property: _release_ver
Private (internal) property: _release_date


Games
=====
{'Wow': {'s_nice_name': 'World of Warcraft'}, 'OW': {'i_price': 35},
'SC2': {'s_responsible': 'Carles Mateo', 'i_price': 50, 'b_is_free':
True}}
```

So this sample server to illustrates that with ****kwargs** we can support different function/method signature and we know the formal parameter name.
I have called the method with more and more parameters each time, everything works like a charm.
I implemented a different action if the parameter starts with underscore _ for demonstration purpose, and those fields will not be added to the final array.

I'm not a big fan of this, I prefer to have my own array of parameters, cause I try to estandardize the way I work when I do in Python, Java, PHP...

```
def test_multiply_ko():
```

When you want to test a method call multiply.
The thing is that by the name you will identify what is being tested.
In the first case we will assert for something that will work, for example:

```
assert multiply(3,5) == 15
```

In the second case we will assert for something that will fail, for example:

```
assert multiply(3,5) != 21
```

Then you can add more sense, like for example testing with decimals.

```
assert multiply(3.1, 5) == 15.5
```

If the test fails you discover that your multiply method was casting to integer.
The default unit testing for Python is unittest library. However py.test is much better, in my opinion.

Installing py.test in Ubuntu[66]

Installing py.test for Python2

```
pip install -U pytest

sudo apt install python-pytest
```

Installing py.test for Python3

```
sudo apt install python3-pip

pip3 install pytest
```

Running the Tests in a Virtual Machine like VirtualBox

As we will program normally for Linux, you should run and test your code in Linux.
You can install Linux on VirtualBox, with Python pytest, and run everything from there.

66 It works for several versions, I use Ubuntu 19.04

In this case I recommend Ubuntu 19.04

Running the Tests in a Docker container

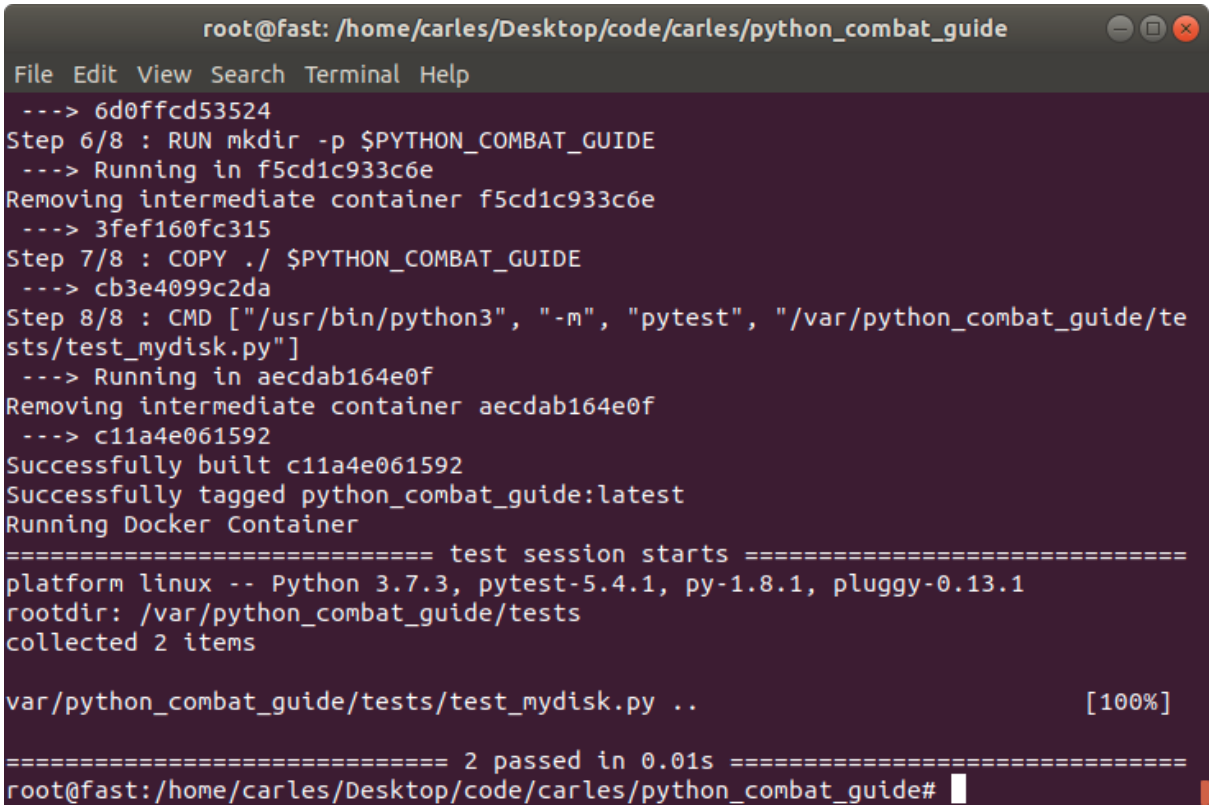
Nowadays, specially with Microservices when we roll a new version of code we generate immutable images where the tests are run, and that same image is tested in different environments (test, stagging, production).
Many companies use CI/CD[67] where a Software like Jenkins will execute a pipeline that will run the tests of the last version of code and deploy to production.
The term immutable images refer to the fact that if a Software bug occur, we don't fix it in the running Docker containers[68], we just roll out a new image and we substitute the old failing Docker container by the new ones.

Also for your convenience, in case you don't want to install all the required packages for pytest in your workstation or if you use Windows, I have provided a Dockerfile that will run the first test, and that you can easily modify to run the other tests.

If you run Docker in a Linux machine or Virtual Machine, you can run the script that I provided and will do everything for you:

```
sudo ./run_docker_image.sh
```

67Continuous Integration Continuous Deployment
68 However, often we will need to investigate on those containers what went wrong.
That's why I like to ship the containers with some tools like: htop, strace, vim, mc…
That few added space is worth when troubleshooting and debugging.



Location of the code samples and tests

You can download the source code samples and the Tests from:
https://gitlab.com/carles.mateo/python_combat_guide

You can clone to you computer with:

```
git clone https://gitlab.com/carles.mateo/python_combat_guide.git
```

Writing the first test

Create a file called test_hello.py
The name is important, cause pytest will consider a test file anything starting by test_ and ending in .py

In this file add the next code:

test_hello.py

```
def hello(s_name=""):
    if s_name == "":
```

```
        return "Hello World!"

    return "Hello " + s_name

def test_hello():
    s_result = hello("Michela")
    assert s_result == "Hello Michela"

    s_result = hello()
    assert s_result == "Hello World!"
```

Now, from the same directory **test_hello.py** exists, run:
`pytest`

Please note that in this sample we mixed code tests and code in the same file for simplicity.

Using classes for testing

Once you get used, you will be very quickly writing test files as classes.

Then your test_ file will contain a class called TestSomething:

For our previous example, we could have a class called `TestHello`:

test_hello2.py

```
class Hello:

   def hello(self, s_name=""):
       """Returns Hello yourname or Hello World!
       :return: str: Hello yourname
       """
       if s_name == "":
           return "Hello World!"

       return "Hello " + s_name


class TestHello:

   def test_hello(self):
       o_hello = Hello()

       s_result = o_hello.hello("Michela")
       assert s_result == "Hello Michela"

       s_result = o_hello.hello()
       assert s_result == "Hello World!"
```