Notebook 1: Python Overview

Motivations

Spark provides multiple *Application Programming Interfaces* (API), i.e. the interface allowing the user to interact with the application. The main APIs are Scala and Java APIs, as Spark is implemented in Scala and runs on the Java Virtual Machine (JVM). Since the 0.7.0 version, a Python API is available, also known as PySpark. An R API has been released with 1.5.0 version. During this course, you will be using Spark 2.4.4.

Throughout this course we will use the Python API for the following reasons:

- R API is still too young and limited to be relied on. Besides, R can quickly become a living hell when using immature libraries.
- Many of you are wanabee datascientists, and Python is a must-know language in data industry.
- Scala and Java APIs would have been quite hard to learn given the length of the course and your actual programming skills.
- Python is easy to learn, and even easier if you are already familiar with R.

The goal of this session is to teach (or remind) you the syntax of basic operations, control structures and declarations in Python that will be useful as datascientist. Keep in mind that we do not have a lot of time, and that you should be able to create functions and classes and to manipulate them at the end of the lab. If you don't get that, the rest of the course will be hard to follow. Don't hesitate to ask for explanations and/or more exercises if you don't feel confident enough at the end of the lab.

Note: Python comes in two flavours, Python 2 and Python 3. Python 2 is now officially deprecated so there is no longer any support. However, a lot of companies have still a lot of code in Python 2 (Tensorflow only supports python 3 since 2019 and Google is well known for his extensive use of Python 2). In this course, we will use Python 3 rather than Python 2. Note that if you know Python 2, learning Python 3 will be lightning-fast. For those who are interested, you can quickly learn Python 3 syntax over here or find some cheatsheets highlighting the differences between the two versions. Spark Python API is compatible with Python 3 since Spark 1.4.0.

When you look at stackoverflow, please be aware that they might share snippets of code in another python version and that you might need to translate them. So no stupid copy-paste.

This introduction relies on Learn Python in Y minutes

Introduction

Python is a high level, general-purpose interpreted language. Python is meant to be very concise and readable, it is thus a very pleasant language to work with.

1. Primitive Datatypes and Operators

Read section 1 of Learn python in Y Minutes (if you already know Python, you can skip this step). Then, replace `???` in the following cells with your code to answer the questions. To get started, please run the following cell.

In [ ]:

```
```

Compute 4 + 8

In [ ]:

```
4+8
```

Out[ ]: `12`

Compute 4 * 8

In [ ]:

```
4*8
```

Out[ ]: `32`

Compute 4 / 8 (using the regular division operation, not integer division)

In [ ]:

```
4.0/8.0
```

Out[ ]: `0.5`

Compute 4^8

In [ ]:

```
4**8
```

Out[ ]: `65536`

Check if the variable `foo` is None:

In [ ]:

```
foo = None
foo == None
```

Out[ ]: `True`

2. Variables and Collections

Same as before, read the corresponding section, and answer the questions below.

Now you're asked to `print` your results instead of just output them.

Please always remember that there is a difference between the output of your code (which is the result of the last executed line) and the printed output.

In [ ]:

```
txt = "This is a text"
print(txt)
new_txt = "This is another text" # assignment doesn't return anything
print(txt) # return the value of txt
print(new_txt) # return the value of new_txt
```

```
This is a text
This is a text
This is another text
```

In [ ]:

```
# Declare a variable containing a float of your choice and print it
# From now on, when you will be asked to print something, please use the print statement.
my_var = 16.18
print(v)
```

```
16.18
```

In [ ]:

```
# Create a list containing strings and store it in a variable
my_list = ["a", "b", "abc"]
# Append a new string to this list
my_list.append("newstring")
# Append an integer to this list and print it
my_list.append(16)
print(my_list)
```

```
['a', 'b', 'abc', 'newstring', 16]
```

Note that the modifications on list objects are performed inplace, i.e.

```
li = [1, 2, 3]
li.append(4)
li  # => [1, 2, 3, 4]
```

In [ ]:

```
# Mixing types inside a list object can be a bad idea depending on the situation.
# Remove the integer you just inserted in the list and print it
my_list.remove(16)
print(my_list)
```

```
['a', 'b', 'abc', 'newstring']
```

In [ ]:

```
# Print the second element of the list
print(my_list[1])
```

```
b
```

In [ ]:

```
x = 42.567
# Print x with only two decimal (rounding)
print(round(x,2))
# Print x with only two decimal (trunking)
print(str(x)[:5])
```

```
42.57
42.56
```

You can access list elements in reverse order, e.g.

```
li[-1]  # returns the last element of the list
li[-2]  # returns the second last element of the list
```

and so on...

In [ ]:

```
# Extend your list with new_list and print it
new_list = ["We", "are", "the", "knights", "who", "say", "Ni", "!"]
my_new_list = my_list + new_list
print(my_new_list)
```

```
['a', 'b', 'abc', 'newstring', 'We', 'are', 'the', 'knights', 'who', 'say', 'Ni', '!']
```

In [ ]:

```
# Replace "Ni" by "Ekke Ekke Ekke Ekke Ptang Zoo Boing" in the list and print it
my_new_list[my_new_list.index('Ni')] = 'Ekke Ekke Ekke Ekke Ptang Zoo Boing'
print(my_new_list)
```

```
['a', 'b', 'abc', 'newstring', 'We', 'are', 'the', 'knights', 'who', 'say', 'Ekke Ekke Ekke Ekke Ptan
g Zoo Boing', '!']
```

In [ ]:

```
# Compute the length of the list and print it
mnl_len = len(my_new_list)
print(mnl_len)
```

```
12
```

In [ ]:

```
# What is the difference between lists and tuples?
"""
Lists are mutable while tuples are immutable ; i.e. cannot access and modify elements of the latter.
"""
```

In [ ]:

```
# Create a dictionary containing the following mapping:
# "one" : 1
# "two" : 2
# etc. until you reach "five" : 5
baz = {
    "one" : 1,
    "two" : 2,
    "three" : 3,
    "four" : 4,
    "five" : 5
}
```

```
{'four': 4, 'three': 3, 'five': 5, 'two': 2, 'one': 1}
```

In [ ]:

```
# Check if the key "four" is contained in the dict
# If four is contained in the dict, print the associated value
print('four' in baz)
print(baz["four"])
```

```
True
4
```

In [ ]:

```
gibberish = list("fqfgsrhrfeqluihjgrshioprqoqeionfvnorfiqeo")
# Find all the unique letters contained in gibberish. Your answer should fit in one line of code
unique_letters = set(gibberish)
print(unique_letters)
```

```
set(['e', 'g', 'f', 'i', 'h', 'j', 'l', 'o', 'n', 'q', 'p', 's', 'r', 'u', 'v'])
```

You should now be able to answer the following problem using dictionaries, lists and sets. Imagine you owe money to your friends because your forgot your credit card last time you went out for drinks. You want to remember how much you owe to each of them in order to refund them later. Which data structure would be useful to store this information? Use this data structure and fill it in with some debt data in the cell below:

In [ ]:

```
debts = {
    "Asterix" : 5,
    "Obelix" : 10
}
```

Another party night with more people, yet you forgot your credit card again... You meet new friends who buy you drinks. Create another data structure as above with different data, i.e. include friends that were not here during the first party and new friends.

In [ ]:

```
debts_2 = {
    "Asterix" : 12,
    "Panoramix" : 3,
    "Assurancetourix" : 106
}
```

Count the number of new friends you made that second night. Print the name of the friends who bought you drinks during the second party, but not during the first.

In [ ]:

```
new_friends = set(debts_2) - set(debts) # should fit in one line
nb_new_friends = len(new_friends) # should fit in one line
print(new_friends)
print(nb_new_friends)
```

```
set(['Assurancetourix', 'Panoramix'])
2
```

3. Control flow

Same as before, read the corresponding section, and answer the questions below. You can skip the paragraph on exceptions for now.

In [ ]:

```
# Code the following:
# if you have made more than 5 friends that second night,
# print "Yay! I'm super popular!", else, print "Duh..."
if nb_new_friends > 5 :
  print("Yay! I'm super popular!")
else :
  print("Duh...")
```

```
Duh...
```

In [ ]:

```
# Now, thank each new friend iteratively, i.e.
# print "Thanks <name of the friend>!" using loops and string formatting (cf. section 1)
for friends in new_friends :
  print("Thanks " + friends + "!")
```

```
Thanks Assurancetourix!
Thanks Panoramix!
```

In [ ]:

```
# Sum all the number from 0 to 15 (included) using what we've seen so far (i.e. without the function su
m() )
sum_to_fifteen = 0
for i in range(15+1):
  sum_to_fifteen = sum_to_fifteen + i
print(sum_to_fifteen)
```

```
120
```

In [ ]:

```
# Note: you can break a loop with the break statement
for i in range(136):
    print i
    if i >= 2:
        break
```

```
0
1
2
```

In [ ]:

```
# enumerate function can be very useful when dealing with iterators:
for i, value in enumerate(["a", "b", "c"]):
    print(value, i)
```

```
('a', 0)
('b', 1)
('c', 2)
```

4. Functions

Things are becoming more interesting. Read section 4. It's ok if you don't get the args/kwargs part. Be sure to understand basic function declaration and anonymous function declaration. Higher order functions, maps, and filters will be covered during the next lab.

Write a Python function that checks whether a passed string is palindrome or not. Note: a palindrome is a word, phrase, or sequence that reads the same backward and forward, e.g. "madam" or "nurses run". Hint: strings are lists of characters e.g.

```
a = "abcdef"
a[2] => c
```

If needed, here are some tips about string manipulation.

In [ ]:

```
def isPalindrome(string_input):
    string_input = string_input.replace(" ", "") # remove spaces
    rev = string_input[::-1]
    if rev == string_input :
      return("True")
    else :
      return("False")

print(isPalindrome('aza'))         # Simple palindrome
print(isPalindrome('nurses run')) # Palindrome containing a space
print(isPalindrome('palindrome')) # Not a palindrome
```

```
True
True
False
```

Write a Python function to check whether a string is pangram or not. Note: pangrams are words or sentences containing every letter of the alphabet at least once. For example: "The quick brown fox jumps over the lazy dog".

Hint

In [ ]:

```
import string

# In this function, "alphabet" argument has a default value: string.ascii_lowercase
# string.ascii_lowercase contains all the letters in lowercase.
def ispangram(string_input, alphabet=string.ascii_lowercase):
  if set(string_input) >= set(alphabet) :
    return("True")
  else :
    return("False")

print(ispangram('The quick brown fox jumps over the lazy dog'))
print(ispangram('The quick red fox jumps over the lazy dog'))
```

```
True
False
```

Python lambda expressions

When evaluated, lambda expressions return an anonymous function, i.e. a function that is not bound to any variable (hence the "anonymous"). However, it is possible to assign the function to a variable. Lambda expressions are particularly useful when you need to pass a simple function into another function. To create lambda functions, we use the following syntax

```
lambda argument1, argument2, argument3, etc. : body_of_the_function
```

For example, a function which takes a number and returns its square would be

```
lambda x: x**2
```

A function that takes two numbers and returns their sum:

```
lambda x, y: x + y
```

`lambda` generates a function and returns it, while `def` generates a function and assigns it to a name. The function returned by `lambda` also automatically returns the value of its expression statement, which reduces the amount of code that needs to be written.

Here are some additional references that explain lambdas: Lambda Functions, Lambda Tutorial, and Python Functions.

Here is an example:

In [ ]:

```
# Function declaration using def
def add_s(x):
    return x + 's'

print type(add_s)
print add_s
print add_s('dog')
```

In [ ]:

```
# Same function declared as a lambda
add_s_lambda = lambda x: x + 's'
print type(add_s_lambda)
print add_s_lambda  # Note that the function shows its name as <lambda>
print add_s_lambda('dog')
```

In [ ]:

```
# Code a function using a lambda expression which takes
# a number and returns this number multiplied by two.
multiply_by_two = lambda x : 2*x
print multiply_by_two(5)

print(multiply_by_two(10) == 20)
```

```
10
True
```

Observe the behavior of the following code:

In [ ]:

```
def add(x, y):
    return(x+y)

def sub(x, y):
    return(y-x)

functions = [add, sub]
print(functions[0](1, 2))
print(functions[1](3, 4))
```

```
3
1
```

Code the same functionality, using lambda expressions:

In [ ]:

```
lambda_functions = [lambda x,y : x+y ,  lambda x,y : y-x]

print(lambda_functions[0](1, 2) == 3)
print(lambda_functions[1](3, 4) == -1)
```

```
True
False
```

Lambda expressions can be used to generate functions that take in zero or more parameters. The syntax for `lambda` allows for multiple ways to define the same function. For example, we might want to create a function that takes in a single parameter, where the parameter is a tuple consisting of two values, and the function adds the two values. The syntax could be either

```
lambda x: x[0] + x[1]
```

or

```
lambda (x, y): x + y
```

If we called either function on the tuple `(1, 2)` it would return `3`.

In [ ]:

```
# Example:
add_two_1 = lambda x, y: (x[0] + y[0], x[1] + y[1])
add_two_2 = lambda (x0, x1), (y0, y1): (x0 + y0, x1 + y1)
print('add_two_1((1,2), (3,4)) = {0}'.format(add_two_1((1,2), (3,4))))
print('add_two_2((1,2), (3,4)) = {0}'.format(add_two_2((1,2), (3,4))))
```

```
add_two_1((1,2), (3,4)) = (4, 6)
add_two_2((1,2), (3,4)) = (4, 6)
```

In [ ]:

```
# Use both syntaxes to create a function that takes in a tuple of three values and reverses their order
# E.g. (1, 2, 3) => (3, 2, 1)
reverse1 = lambda x: (x[2], x[1], x[0])
reverse2 = lambda (x0, x1, x2): (x2, x1, x0)

print(reverse1((1, 2, 3)) == (3, 2, 1))
print(reverse2((1, 2, 3)) == (3, 2, 1))
```

```
True
True
```

Lambda expressions allow you to reduce the size of your code, but they are limited to simple logic. The following Python keywords refer to statements that cannot be used in a lambda expression: `assert`, `pass`, `del`, `print`, `return`, `yield`, `raise`, `break`, `continue`, `import`, `global`, and `exec`. Assignment statements (`=`) and augmented assignment statements (e.g. `+=`) cannot be used either. If more complex logic is necessary, use `def` in place of `lambda`.

5. Classes

Classes allow you to create objects. Object Oriented Programming (OOP) can be a very powerful paradigm. If done well, OOP allows you to improve the modularity and reusability of your code, but that's the subject of an entire other course. Here is a *very* short introduction to it.

By convention, class names are written in camel case, e.g. `MyBeautifulClass`, while variable and function names are written in snake case, e.g. `my_variable`, `my_very_complex_function`

Classes contain methods (i.e. functions owned by the class) and attributes (i.e. variables owned by the class). When you define a class, first thing to do is to define a specific method, the constructor. In Python, the constructor is called `__init__`. This method is used to create the instances of an object. Example:

```
class MyClass:

    def __init__(self, first_attribute, second_attribute):
        self.first_attribute = first_attribute
        self.second_attribute = second_attribute
```

This class has two attributes, and one (hidden) method, the constructor. To create an instance of this class, one simply does:

```
instance_example = MyClass(1, "foo")
```

Then, the attributes can easily be accessed to:

```
instance_example.first_attribute  # => 1
instance_example.first_attribute  # => "foo"
```

In [ ]:

```
# Run this example
class MyClass:

    def __init__(self, first_attribute, second_attribute):
        self.first_attribute = first_attribute
        self.second_attribute = second_attribute

instance_example = MyClass(1, "foo")
print(instance_example.first_attribute)
instance_example.__init__(3,4)  # In real life, it is rare to reinit an object.
print(instance_example.first_attribute)
```

```
1
3
```

`self` denotes the object itself. When you declare a method, you have to pass `self` as the first argument of the method:

class MyClass:

```
def __init__(self, first_attribute, second_attribute):
    self.first_attribute = first_attribute
    self.second_attribute = second_attribute

def method_baz(self):
    print "Hello! I'm a method! I have two attributes, initialized with values %s, %s"%(self.fir
st_attribute, self.second_attribute)
```

indeed, when we call

```
instance_example = MyClass(1, "foo")
instance_example.method_baz()
```

the `self` object is implicitely passed to `method_baz` as an argument. Think of the method call as the following function call

```
method_baz(instance_example)
```

In [ ]:

```
# Run this example
class MyClass:

    def __init__(self, first_attribute, second_attribute):
        self.first_attribute = first_attribute
        self.second_attribute = second_attribute

    def class_method(self):
        print("Hello! I'm a method! My class has two attributes, of value {0}, {1}".format(self.first_a
ttribute, self.second_attribute))

instance_example = MyClass(1, "foo")
# Call to a class method
instance_example.class_method()
```

```
Hello! I'm a method! My class has two attributes, of value 1, foo
```

Now, the tricky part. You can declare **static** methods, i.e. methods that don't need to access the data contained in `self` to work properly. Such methods do not require the `self` argument as they do not use any instance data. They are implemented in the following way:

In [ ]:

```
# Run this example
class MyClass:

    def __init__(self, first_attribute, second_attribute):
        self.first_attribute = first_attribute
        self.second_attribute = second_attribute

    def class_method(self):
        print("Hello! I'm a method! My class has two attributes, of value {0}, {1}".format(self.first_a
ttribute, self.second_attribute))

    @staticmethod
    def static_method():
        print("I'm a static method!")

instance_example = MyClass(1, "foo")
# Call to a class method
instance_example.class_method()
# Call to a static method
instance_example.static_method()
```

```
Hello! I'm a method! My class has two attributes, of value 1, foo
I'm a static method!
```

In [ ]:

```
# Call to a static method without class instanciation
MyClass.static_method()
```

```
I'm a static method!
```

In [ ]:

```
# Call to a class method without class instanciation: raises an error
# MyClass.class_method()
# => TypeError: unbound method class_method() must be called with MyClass instance as first argument (g
ot nothing instead)
```

You can set attributes without passing them to the constructor:

In [ ]:

```
# Run this example
class MyClass:

    default_attribute = 42

    def __init__(self, first_attribute, second_attribute):
        self.first_attribute = first_attribute
        self.second_attribute = second_attribute

    def method_baz(self):
        print("Hello! I'm a method! I have two attributes, initialized with values %s, %s"%(self.first_
attribute, self.second_attribute))

    @staticmethod
    def static_method():
        print("I'm a static method!")

instance_example = MyClass(1, "foo")
print(instance_example.default_attribute)
```

```
42
```

In [ ]:

```
# Write a Python class named Rectangle which is
# constructed by a length and width
# and has two class methods
# - "rectange_area", which computes the area of a rectangle.
# - "rectangle_perimeter", which computes the perimeter of a rectangle.
#
# The Rectangle class should have an attribute n_edges equal to 4
# which should not be initialized by the __init__ constructor.
#
# Declare a static method "talk" that returns "Do you like rectangles?" when called

class Rectangle:

    def __init__(self, length, width):
        self.length = length
        self.width  = width

    def rectangle_area(self):
        return self.length*self.width

    def rectangle_perimeter(self) :
        return 2*self.length+2*self.width

    n_edges = 4

    @staticmethod
    def talk():
        return "Do you like rectangles?"


new_rectangle = Rectangle(12, 10)
print(new_rectangle.rectangle_area() == 120)# rectangle_area method
print(new_rectangle.rectangle_perimeter() == 44) # rectangle_area method
print(Rectangle.n_edges == 4) # constant attibute
print(Rectangle.talk() == "Do you like rectangles?") # Rectangle talk static method
```

```
True
True
True
True
```

In machine learning, when you're manipulating large images, you often need to break the diwn into smaller, more manageable images.

- Modify the previous rectangle class to include a list of patches.
- Write a Python class named Patch which is a square of size 1x1. Each patch should be identified by his coordinates.
- Use the __iter__ function to create a get_next_patch() in the Rectangle class.

In [3]:

```
class Patch :

    def __init__(self, coord_x, coord_y):
      self.coord_x = coord_x
      self.coord_y = coord_y

    def coord(self) : # Start from bottom left of the 1x1 square
      coord = [(self.coord_x,self.coord_y),
               (self.coord_x,self.coord_y+1),
               (self.coord_x+1,self.coord_y+1),
               (self.coord_x+1,self.coord_y)]
      return coord

class Rectangle:

    def __init__(self, length, width):
        self.length = length
        self.width  = width
        self.idx = -1

    def rectangle_area(self):
        return self.length*self.width

    def rectangle_perimeter(self) :
        return 2*self.length+2*self.width

    n_edges = 4

    @staticmethod
    def talk():
        return "Do you like rectangles?"

    def list_of_patches(self) :
      patches = [] # store patches coordinates
      for i in range(0, self.length):
        for j in range(0, self.width):
          patches.append(Patch(i,j).coord()) # call Patch class coord function
      return patches

    def __iter__(self) :
      return self

    def __next__(self) : # use only next because Python 2 / Python 3 incompatibility...
        self.idx += 1
        if self.idx < self.length*self.width :
            self.patch = Rectangle(self.length, self.width).list_of_patches()[self.idx]
            return self.patch
        raise StopIteration

print(Rectangle(2,3).list_of_patches())

for patch in Rectangle(2, 3):
  print(patch)
```

```
[[(0, 0), (0, 1), (1, 1), (1, 0)], [(0, 1), (0, 2), (1, 2), (1, 1)], [(0, 2), (0, 3), (1, 3), (1,
2)], [(1, 0), (1, 1), (2, 1), (2, 0)], [(1, 1), (1, 2), (2, 2), (2, 1)], [(1, 2), (1, 3), (2, 3), (2,
2)]]
[(0, 0), (0, 1), (1, 1), (1, 0)]
[(0, 1), (0, 2), (1, 2), (1, 1)]
[(0, 2), (0, 3), (1, 3), (1, 2)]
[(1, 0), (1, 1), (2, 1), (2, 0)]
[(1, 1), (1, 2), (2, 2), (2, 1)]
[(1, 2), (1, 3), (2, 3), (2, 2)]
```

Use the same approach to create two classes :

- Sentence which is a list of words
- Word which encapsulate a string with a specific order within the sentence Implement both __str__ and __repr__ for both classes. `__str__` should be readable when `__repr__` should offer you information to helps you debug. In Natural Language Processing you'll use a lot of n-grams (which is the set of *n* sequential words). So you should implement a get_next_bigrams() method in the Sentence class. For instance give the sentence "My name is Brian", it should return an element of the list of bigrams `["My name", "name is", "is Brian"]`

In [7]:

```
class Sentence :

    def __init__(self, sentence):
      self.sentence = sentence
      self.idx = 0

    def __str__(self):
      return self.sentence.split()

    def __repr__(self):
      return '"{}".split()'.format(self.sentence)

    def __iter__(self) :
      return self

    def __next__(self) :
        self.idx += 1
        if self.idx < len(Sentence(self.sentence).__str__()) :
            self.bigram = Sentence(self.sentence).__str__()[self.idx-1:self.idx+1]
            return self.bigram
        raise StopIteration

class Word :
    def __init__(self, word):
      self.word = word

    def __str__(self, sentence, position) :
      l = Sentence(sentence).__str__()
      l.insert(position, self.word) # insert word into list
      return ' '.join(l)  # return back sentence

    def __repr__(self, sentence, position) :
      return '"{}".insert({},{})'.format(sentence, position, self.word)

print(Sentence("This is a test").__str__())
print(Sentence("This is a test").__repr__())

print(Word("dog").__str__('I like', 2))
print(Word("dog").__repr__('I like', 2))

for bigram in Sentence('My name is Brian'):
  print(bigram)
```

```
['This', 'is', 'a', 'test']
"This is a test".split()
I like dog
"I like".insert(2,dog)
['My', 'name']
['name', 'is']
['is', 'Brian']
```

6. Modules

Finally one really important part of python is the ability to create modules. Read https://docs.python.org/fr/3/tutorial/modules.html and create a module named geo with the classes Rectangle & Patch. Import the `geo` module and demonstrate the different calls.

Repeat the operation with an `NLP` module to encapsulate the classes Sentence and Word. In Natural Language Processing you'll use a lot of n-grams (which is the set of *n* sequential words). So you should implement a `get_bigrams(sentence)` method who will return a list of bigrams.

In [4]:
```python
import geo

rectangle = Rectangle(3,2)
print(rectangle.rectangle_area())
print(rectangle.rectangle_perimeter())
print(rectangle.list_of_patches())

patch = Patch(2,2)
print(patch.coord())
```

```
6
10
[[(0, 0), (0, 1), (1, 1), (1, 0)], [(0, 1), (0, 2), (1, 2), (1, 1)], [(1, 0), (1, 1), (2, 1), (2, 0)], [(1, 1), (1, 2), (2, 2), (2, 1)], [(2, 0), (2, 1), (3, 1), (3, 0)], [(2, 1), (2, 2), (3, 2), (3, 1)]]
[(2, 2), (2, 3), (3, 3), (3, 2)]
```

In [12]:
```python
import NLP

test_sentence = Sentence("My name is Brian")

print(test_sentence.__str__())
print(test_sentence.__repr__())
for bigram in test_sentence:
  print(bigram)
```

```
['My', 'name', 'is', 'Brian']
"My name is Brian".split()
['My', 'name']
['name', 'is']
['is', 'Brian']
```

7. Coding style

You'll often ask yourself about which letters should be capital or if you should use _ or other characters.

To help you decide, python comes with PEP-8 which is a set of recommandation.

It is highly advised to read PEP-8 and to apply the rules whenever it's applicable.

You'll get point for style in your code (I mean, you will lose points if the code is not readable).

Fortunately, [the rules of PEP-8 are not perfect and should not always be followed].

In [ ]:
```python
# You should create a github account before the next session :
student_1_github_login = TheoDecreux
student_2_github_login = IliesHachemi
```

Congratulations, you've reched the end of this notebook. =)